



UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH



MASTER THESIS

Structured Flight Plan Interpreter for Drones in AirSim

Francesco Rose

SUPERVISED BY

Cristina Barrado Muxi

**Universitat Politècnica de Catalunya
Master in Aerospace Science & Technology
January 2020**

This Page Intentionally Left Blank

Structured Flight Plan Interpreter for Drones in AirSim

BY

Francesco Rose

DIPLOMA THESIS FOR DEGREE

Master in Aerospace Science and Technology

AT

Universitat Politècnica de Catalunya

SUPERVISED BY:

Cristina Barrado Muxi

Computer Architecture Department

*“E come i gru van cantando lor lai,
faccendo in aere di sé lunga riga,
così vid’io venir, traendo guai,
ombre portate da la detta briga”*

Dante, Inferno, V Canto

This Page Intentionally Left Blank

ABSTRACT

Nowadays, several Flight Plans for drones are planned and managed taking advantages of Extensible Markup Language (XML). In the mean time, to test drones performances as well as their behavior, simulators usefulness has been increasingly growing. Hence, what it takes to make a simulator capable of receiving commands from an XML file is a dynamic interface.

The main objectives of this master thesis are basically three. First of all, the handwriting of an XML flight plan (FP) compatible with the simulator environment chosen. Then, the creation of a dynamic interface that can read whatever XML FP and that will transmit commands to the drone. Finally, using the simulator, it will be possible to test both interface and flight plan.

Moreover, a dynamic interface aimed at managing two or more drones in parallel has been built and implemented as extra objective of this master thesis. In addition, assuming that two drones will be used to test this interface, it is required the handwriting of two more FPs.

In order to achieve all the goals of this project, it has been chosen AirSim as drone-simulator and Python as programming-language for the development of the dynamic interfaces. Python and AirSim can “talk” to each other thanks to the really good list of APIs (Application Programming Interface) provided by the AirSim library for Python.

On the other hand, to write the XML FPs, I took advantages of the RAISE+ documentation (simulator for fixed and rotary wing aircrafts) for building a flight plan (see [10]). I implemented a total of six FPs: two FPs to test the interface for the single drone and four FPs to test the multiple-drones interface (two FPs for each drone). Each pair of FPs has the same path; one uses Geographical coordinates (latitude, longitude, altitude), the other one uses AirSim’s NED coordinates (north, east, down). Since take off and landing are obtained through two Python APIs for AirSim, the flight plan will concern only the mission waypoints.

In the end, I obtained two dynamic interfaces with a high degree of independence from any XML flight plan and AirSim environment chosen. The only requirement is that the FP waypoints have to be compatible with the simulator environment. Moreover, the FP has been created involving four out of all the possible legs that describe drone maneuvers and it has been planned for the Neighborhood AirSim environment. All the limitations will be further discussed in the “Recommendations” section (6.3).

All the topics will be deeply analyzed and successively explained along the master thesis, highlighting the most important features and the problem-solving methodology carried on during the whole project.

This Page Intentionally Left Blank

ACKNOWLEDGEMENTS

I am deeply grateful to my master thesis supervisor Professor Cristina Barrado Muxi for providing me with professional guidance and clear directive during the development of the project.

I am greatly indebted to my flatmate Riccardo Salis and to my colleague Thomas Fili. Their moral and work support is incomparable; I will always bring this great experience with me.

Special appreciation to PhD Ender Çetin for providing me with assistance and support during the development of the project.

I would like to thank my parents and my whole family for supporting me during the master course as well as during all my life. I know I would never reach this goal without them. I owe everything I am to my family.

I am deeply grateful to my lovely girlfriend Cristiana. Her support and patience as well as her comprehension played a key role during the entire master course. I love you.

I would like to thank my Master coordinator Professor Ricard Gonzalez Cinca for providing advice and support during the whole master course. Last but not least, many thanks to the UPC University for all the resources made available.

This Page Intentionally Left Blank

Table of Contents

INTRODUCTION	5
CHAPTER 1 GENERAL DEVELOPMENT OF THE MASTER THESIS	7
1.1. Step 1: XML FPs implementation for Single-Drone Interface	7
1.2. Step 2: Python Interface implementation for Signle Drone	7
1.3. Step 3: AirSim Test	7
1.4. Step 4: Problem Solving Approach	8
1.5. Step 5: XML FPs implementation for Multiple-Drones Interface	8
1.6. Step 6: Python Interface implementation for Multiple Drones.....	8
1.7. Step 7: AirSim Test	8
1.8. Step 8: Problem Solving Approach	8
CHAPTER 2 EXTENSIBLE MARKUP LANGUAGE: XML FILE.....	9
2.1. XML General Features	9
2.2. A little bit of history	9
2.3. Characters of an XML file	9
2.3.1. Tag.....	9
2.3.2. Element.....	10
2.3.3. Attribute	10
2.4. How to build an XML Flight Plan	10
2.4.1. General	10
2.4.2. Leg: definition and classification	11
2.5. FPs Strategy	13
2.5.1. Single-Drone Flight Plan	13
2.5.2. Multiple-Drones Flight Plans	16
2.6. XML Processing	16
CHAPTER 3 AIRSIM SIMULATOR	17
3.1. Overview	17
3.2. General Features: Unreal Engine and AirSim	17
3.3. Neighborhood Environment: Reference Frame System	18
3.4. Surveillance Mission.....	20
CHAPTER 4 DEVELOPMENT OF THE FP INTERPRETER.....	21
4.1. Python features	21
4.2. A little bit of history	21
4.3. Single-Drone Interface.....	21
4.3.1. AirSim	21
4.3.2. XML Parsing	22
4.3.3. Flight Plan Processing	23
4.3.4. Processing of Flight Plan Legs	24
4.3.5. Flight Recording.....	25

4.3.6.	Visualization of Flight Recording	26
4.3.7.	Execution of the FP	26
4.3.8.	Libraries Used.....	26
4.3.9.	Other Functions	27
4.4.	Multiple-Drones Interface	28
4.4.1.	Threading.....	28
4.4.2.	Libraries Used.....	29
4.4.3.	AirSim	29
4.4.4.	Execution of the FP	29
CHAPTER 5 PLOTS AND RESULTS.....		31
5.1.	Plots and Data First Choice Intersection NED FP	31
5.2.	Plots and Data Second Choice Intersection NED FP.....	34
5.3.	Plots and Data Third Choice Intersection NED FP.....	37
5.4.	Plots and Data First Choice Intersection Geographical FP	40
5.5.	Plots and Data Second Choice Intersection Geographical FP	43
5.6.	Plots and Data Third Choice Intersection Geographical FP	46
5.7.	General Comments and Results.....	49
CHAPTER 6 CONCLUSIONS AND RECOMMENDATIONS		51
6.1.	Preliminary Conclusions	51
6.2.	Area for further studies	51
6.3.	Recommendations	52
6.3.1.	Recommendations for both interfaces	52
6.3.2.	Recommendations for Single-Drone interface.....	52
6.3.3.	Recommendations for Multiple-Drones interface.....	53
CHAPTER 7 BIBLIOGRAPHY.....		55
ANNEX		57

List of Figures

Figure 2.1 Locale Settings	19
Figure 2.2 MainFP	20
Figure 2.3 To Fix Leg.....	20
Figure 2.4 Intersection Leg	21
Figure 2.5 Scan Leg.....	21
Figure 2.6 Iterative Leg	22
Figure 2.7 Example of To Fix Leg.....	23
Figure 2.8 Heading of Intersection Leg.....	23
Figure 2.9 Example of Parametric Leg	23
Figure 2.10 Example of Iterative Leg.....	24
Figure 3.1 Car in City Environment – Drone in Neighborhood Environment	27
Figure 3.2 Neighborhood Reference Frame System	27
Figure 3.3 Set default starting point coordinates in “Setting.json”	28
Figure 4.1 Connection to the simulator code lines.....	31
Figure 4.2 Take off code lines.....	31
Figure 4.3 Landing code lines.....	31
Figure 4.4 XML Parsing code lines	32
Figure 4.5 To Fix Leg Processing code lines.....	33
Figure 4.6 Spread Sheet Creation code lines.....	34
Figure 4.7 Spread Sheet Closure code line.....	35
Figure 4.8 LOG variable code line	35
Figure 4.9 Libraries Used code lines	36
Figure 4.10 Get_data Function code lines.....	37
Figure 4.11 Libraries Used code lines	38
Figure 5.1 Theoretical Path – Simulated Path comparison	40
Figure 5.2 Theoretical Coordinates – Simulated Coordinates comparison	41
Figure 5.3 Theoretical Time – Simulated Time comparison	42
Figure 5.4 Theoretical Path – Simulated Path comparison	43
Figure 5.5 Theoretical Coordinates – Simulated Coordinates comparison	44
Figure 5.6 Theoretical Time – Simulated Time comparison	45
Figure 5.7 Theoretical Path – Simulated Path comparison	46
Figure 5.8 Theoretical Coordinates – Simulated Coordinates comparison	47
Figure 5.9 Theoretical Time – Simulated Time comparison	48
Figure 5.10 Theoretical Path – Simulated Path comparison	49
Figure 5.11 Theoretical Coordinates – Simulated Coordinates comparison	50
Figure 5.12 Theoretical Time – Simulated Time comparison	51

Figure 5.13 Theoretical Path – Simulated Path comparison	52
Figure 5.14 Theoretical Coordinates – Simulated Coordinates comparison	53
Figure 5.15 Theoretical Time – Simulated Time comparison	54
Figure 5.16 Theoretical Path – Simulated Path comparison	55
Figure 5.17 Theoretical Coordinates – Simulated Coordinates comparison	56
Figure 5.18 Theoretical Time – Simulated Time comparison	57
Figure 6.1 Setting.json for Single-Drone Interface.....	61
Figure 6.2 Setting.json for Multiple-Drones Interface.....	62

List of tables

Table 2-1 N0 and N1 waypoints coordinates	22
Table 2-2 Home1 and Home2 waypoints coordinates	25
Table 3-1 Default Starting Point Coordinates – Conversion Parameters	28
Table 5-1 Theoretical Time Values - Simulated Time Values comparison	41
Table 5-2 Theoretical Time Values - Simulated Time Values comparison	44
Table 5-3 Theoretical Time Values - Simulated Time Values comparison	47
Table 5-4 Theoretical Time Values - Simulated Time Values comparison	50
Table 5-5 Theoretical Time Values - Simulated Time Values comparison	53
Table 5-6 Theoretical Time Values - Simulated Time Values comparison	56
Table 5-7 Simulated Time Values comparison of the same path	57

INTRODUCTION

Drones are becoming more and more an active part of our life. This technology is useful for a wide range of application field like panoramic scan, monitoring and surveillance, low-altitude photograph as well as cargo delivery, agriculture, etc.

On the other hand, several issues are directly linked to the drone world. In the “Area for further studies” section (6.2) I will present all these issues that have to be taken into account to fly a real drone in a populated area. The first problem that has to be analyzed is the respect for privacy. Then, it has to be considered that this technology is “easy” to be used: thieves, terrorists, arsonists and thousands of other evil-minded people will badly take advantages of drones to reach their purpose. Hence, the need for “police-drones” that will protect us catching the “evil-drones”.

Currently, humans pilot most of the drones and they introduce all the instructions using telecommands or drone ground stations. Moreover, automated drones are already on market but their autopilot allows only easy movements. The next step is to implement an autopilot that allows more difficult maneuvers.

This master thesis will be focused on the building of two dynamic interfaces that will allow the final customer to upload whatever “Extensible-Markup-Language Flight-Plan” (XML FP) on drones in the simulator environment. Then, waypoints informations will be translated into commands for the drones in order to follow the required path. After that, drones will be able to follow the flight plan without any external help to succeed take-off, mission and landing.

In order to test both my interfaces and FPs, I created a use case: Neighborhood-Surveillance-Mission (NSM) is aimed to successfully monitor the chosen AirSim Environment and detect possible thieves. With respect to the single-drone interface, my drone *Carlo* has been commissioned to monitor the Neighborhood through three different scanning paths. *Paolo e Francesca* will carry out the same mission, but they will be operated by the multiple-drones interface. Later on in the master thesis, my FPs and the strategies I adopted to reach the objectives of this work will be deeply treated.

Below, the structure of the doc is presented. Chapter 1 points out the methodology followed during the development of the project. Chapters 2 and 3 deal with two of the tools used in this master thesis (XML and AirSim), giving a quick look at both their general features and history. Moreover, it is explained how and when I used them, underlining the important aspects that have to be taken into account. In Chapter 4, Python, its history and its usage are deeply analyzed. I will clarify how I built my scripts, explaining all the functions and the several parts forming the codes. Chapter 5 contains all the significant plots and data obtained running the single-drone interface. In the last section of this chapter the results are presented. In Chapter 6 I draw the main conclusions of the whole project, treating the issues related with the drone world and the recommendations to be followed to correctly use my dynamic interfaces. Chapter 7 is the bibliography. In Chapter 8 all the images of the six XML FPs and of the two Python codes are presented.

Chapter 1

GENERAL DEVELOPMENT OF THE MASTER THESIS

The goal of this chapter is to put in evidence the procedure followed in order to achieve the objectives of the master thesis. To reach all the requirements of this project, I took an inverse-engineering approach as well as a problem-solving methodology.

1.1. Step 1: XML FPs implementation for Single-Drone Interface

The first goal of this work is the creation of an XML file containing a Flight Plan for *Carlo* inside the AirSim's Neighborhood environment. Hence, the selection of compatible waypoints plays a key role for the correct development of the mission. Python-AirSim's APIs allow us to move the drone around the map while getting information on its position (both Geographical and NED coordinates). After that, I built the FPs according to the RAISE+ documentation (see [10]) taking advantages of the chosen points. In the end, I obtained two flight plans with the same path but with different coordinates, one with Geographical and one with NED (North, East, Down). Since AirSim simulator takes only NED coordinates as input, a conversion of the Geographical coordinates is also required. In sub-section 2.5.1, all the information related to these XML Flight Plans and the strategy I adopted will be shown; the conversion parameters are in section 3.3. Images of both flight plans are in the "Annexes" (Chapter 8).

1.2. Step 2: Python Interface implementation for Single Drone

In order to accomplish the second objective, I took advantage of Python as programming language. First of all, I searched how parse an XML file on Python and how reach and store all the data encoded inside. Then, I created functions both to represent the different legs of the flight plan and to execute it in the correct order. I also implemented lines of code to get different significant plots and to store position as well as time data. The dynamic interface has some limits that will be treated in the "Recommendations" section (6.3), but overall it could be said that the interface does not depend on anything, it automatically works with any kind of flight plan and inside all Airsim environments. In section 4.3, the whole code will be deeply examined to put in evidence the characteristics of all the functions and to easily understand how the dynamic interface has been built. Chapter 5 shows all the plots and the results of the project. Images of all the code in the "Annexes" (Chapter 8).

1.3. Step 3: AirSim Test

The last step is the test of the dynamic interface with the implemented Flight Plan. If the drone successfully completes the flight plan from take off to landing, the dynamic interface and the flight plan are well-written. If the drone stops, falls, breaks, reaches undesired positions, collides with obstacles or cuts the scans, a rework of the xml file or of the dynamic interface or both reworks are required.

1.4. Step 4: Problem Solving Approach

To write the “perfect” code and the “best” flight plan, I encountered a series of problems that I was able to solve taking both a problem-solving methodology and an inverse-engineering approach. I usually write the code on paper to better understand how it works and how I can obtain what I want; then, I improve and fix it directly on the machine.

1.5. Step 5: XML FPs implementation for Multiple-Drones Interface

As well as in Step 1, I built four FPs. *Paolo* will use two of them containing the same path (one with Geographical coordinates, the other one with NED coordinates); *Francesca* will use the other two FPs that contain another path. In sub-section 2.5.2, all the information related to these XML Flight Plans and the strategy I adopted will be shown. Images of the four flight plans are in the “Annexes” (Chapter 8).

1.6. Step 6: Python Interface implementation for Multiple Drones

In order to implement the multiple-drones interface, one theoretical concept more is required. Threading must be used to allow Python’s simultaneous handling of two different drones. In section 4.4, the whole code will be deeply examined to put in evidence the characteristics of the Python’s *class* for threading and its functions. Limits for this interface are treated in the “Recommendations” section (6.3). Images of all the code in the “Annexes” (Chapter 8).

1.7. Step 7: AirSim Test

The last step is the test of the dynamic interface with the implemented Flight Plans. If both drones successfully complete their flight plans from take off to landing, the dynamic interface and the flight plans are well-written. If just one of the drones stops, falls, breaks, reaches undesired positions, collides with obstacles or cuts the scans, a rework of the xml files or of the dynamic interface or both reworks are required.

1.8. Step 8: Problem Solving Approach

To write the “perfect” code and the “best” flight plans, I encountered a series of problems that I was able to solve taking both a problem-solving methodology and an inverse-engineering approach. I usually write the code on paper to better understand how it works and how I can obtain what I want; then, I improve and fix it directly on the machine.

Chapter 2

EXTENSIBLE MARKUP LANGUAGE: XML FILE

2.1. XML General Features

XML stands for “*Extensible Markup Language*” and it allows the encoding of documents through both a set of rules and the definition of elements. The great newness is the encoding format that is “both human-readable and Machine-readable” [21]. XML is a “restricted form of SGML” (Standard Generalized Markup Language) and it has to be completely interoperable with SGML and HTML [5].

This language has been implemented to simplify data sharing and data transport as well as data storage and data availability; XML is well known to be self-descriptive and it easily allows platform changes [22]. Moreover, XML has to support numbers of applications as well as be compatible with SGML (see [5]).

Meanwhile extensible-markup-language started to catching on among the informatic community, programmers developed many APIs (Application Programming Interface) to both read and process XML data [21].

2.2. A little bit of history

In 1996 the World Wide Web Consortium (W3C), “the main international standards organization for the World Wide Web” (see [20]), constituted an “*XML Working Group*” which successively developed the XML language [5].

This working group was headed by Jon Bosak of “Sun Microsystems” who collaborated with Tim Bray and James Clark. Bosak decided that HTML could not be able to satisfy the great information trade required. Exchanging data without their meaning is not enough, the machine will not understand such information. Hence, he focused his attention on SGML language and its power. On the other hand, Clark introduced the name XML and the idea of “self-closing elements” [3].

2.3. Characters of an XML file

XML files are composed by units called entities. These units have storage capability and they can contain parsed or unparsed data. Parsed-data characters are divided in “*data character*” and “*markup character*” [5]; these two objects have different applications depending on different syntactic rules. Markup strings generally begin with “<” and end with “>” (“&” and “;” is another form); then, every string that is not a markup character is a data character or “*content*” [21]. The other three main characters for the implementation of an XML file are: “*Tag*”, “*Element*” and “*Attribute*”.

2.3.1. Tag

“A tag is a markup construct that begins with < and ends with >”. There are three different kind of tags but I only used two out of these for the master thesis: “*start-tag*” (e.g. <stage>) and “*end-tag*” (e.g. </stage>).

2.3.2. Element

An element always “begins with a start-tag and ends with a matching end-tag”. In between, it can be found the “*element’s content*” that can contain markup characters such as other elements called “*child elements*”.

2.3.3. Attribute

A start-tag may be complemented with one or more attributes, a markup construct that associates a value to a name, for the sake of example “<leg id=“zero_point” xsi_type=“TF_Leg”> “. Here, “id” and “xsi_type” are the names of the attributes; “zero_point” and “TF_Leg” are respectively the values.

2.4. How to build an XML Flight Plan

All this section has been written taking advantages of the reference [10]. RAISE+ documentation takes into account RPAS but it is possible to follow the general guideline to implement a flight plan for drones. Hence, the unnecessary parts will be reasonably skipped.

2.4.1. General

In order to design an optimal flight plan, it is mandatory to know how the XML code has to be organized and implemented. The first two childs of the principal root are “*Locale Settings*” and “*MainFP*”.

Locale settings indicates distances, speed and altitude measure units. Moreover, it is indicated the decimal and group separators. Figure 2.1 shows all the possible values for these elements; it also shows an exemple of the “Locale Settings” XML code.

speedUnits		altitudeUnits distanceUnits		decimalSeparator	groupSeparator
ms	m/s	m	meters	in principle it could be any string, but most probably ‘.’ and ‘,’	as in decimalSeparator plus empty
kt	knots	nm	nautical miles		
		ft	feet		

```

<!-- Locale settings -->
<Locale>
  <speedUnits>kt</speedUnits>
  <altitudeUnits>ft</altitudeUnits>
  <distanceUnits>nm</distanceUnits>
  <decimalSeparator>.</decimalSeparator>
  <groupSeparator/>
</Locale>

```

Figure 2.1 Locale Settings

Figure 2.2 is an exemple of the “MainFP” XML code.

```

<MainFP id="FPID">
  <name>Name of the flight plan</name>
  <description>Text describing the flight plan</description>
  <!-- List of stages that form the flight plan follows -->
  <stages> ... </stages>
  <!-- List of whitespace separated emergency plan IDs -->
  <emergency> ... </emergency>
</MainFP>

```

Figure 2.2 MainFP

A drone follows the path contained in the main flight plan; moreover, it has also a name and a description of the mission. Then, a list of all the stages is compiled and they must be executed in the correct order.

I chose to not include an emergency plan since it is not the scope of this project. In the “Area for further studies” section (6.3), the necessity of emergency Flight Plans to fly a real drone will be discussed. Moreover, take-off and landing part are directly performed through python’s APIs for AirSim. This means that the corresponding stages are not implemented in the XML codes.

2.4.2. Leg: definition and classification

Each stage has an identifier and contains all the legs belonging to it. A leg identifies the course to the next point along the flight plan; furthermore, each leg is recognized through its “*xsi_type*” attribute.

A leg can be classified depending on both behavior and functionality. I took into account four types of leg which are the most significant for the purposes of my work.

2.4.2.1. Track to a Fix (TF leg)

This type of leg performs a straight path from waypoint to waypoint and it is identified by the “*xsi_type*=“*fp_TFLeg*” ” attribute. The “*dest*” tag contains as child all the informations related to the point that has to be reached. In addition, “*next*” tag highlights the name of the next waypoints corresponding to another leg. The last leg of the flight plan does not require the next waypoint child.

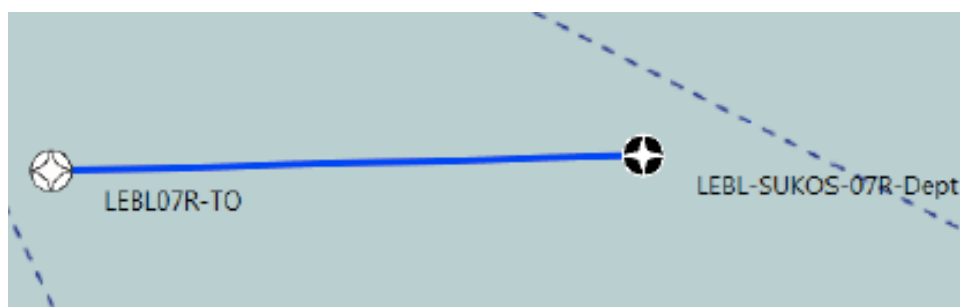


Figure 2.3 To Fix Leg

2.4.2.2. Intersection leg

This kind of leg identifies waypoints where more than one path can be selected depending on a condition that will be chosen by the user. This condition is not inserted along the flight plan. The possible legs to be selected are encoded in the tag “*nextList*” with an unique identifier. The drone will wait hovering until the user chose the needed path. This leg must not be used to emulate an iterative behavior.

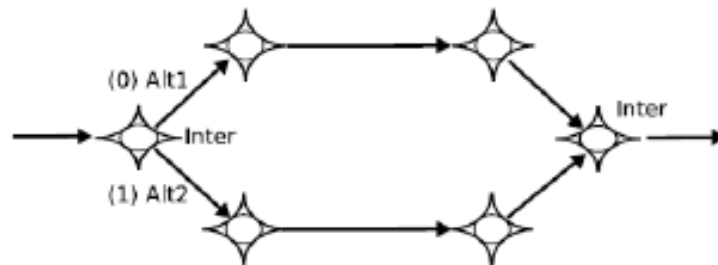


Figure 2.4 Intersection Leg

2.4.2.3. Parametric leg

Parametric legs are very useful, especially for scan paths. Identifying the key parameters, we can chose the corners of an area that will be covered by the scan. “*point1*” tag identifies the entry point of the area and the “*trackseparation*” set the distance between tracks. Then, an operative speed and an operative altitude are required.

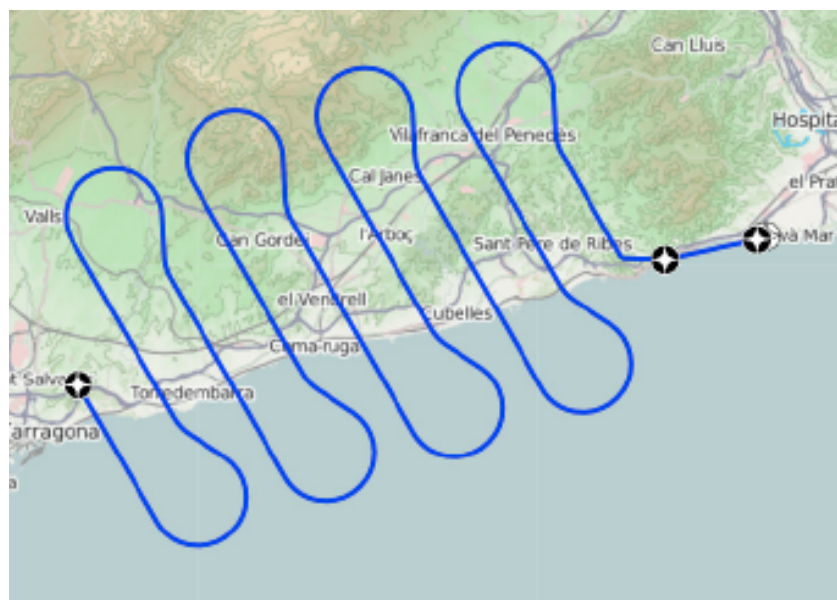


Figure 2.5 Scan Leg

2.4.2.4. Iterative leg

Iterative legs allow users to iterate a sequence of maneuvers a certain number of time. The body contains the legs which have to be iterated and initial as well as final legs

have to be highlighted. Every time the drone performs the last leg, an iteration counter will be incremented. Once the number written in “*UpperBound*” tag is reached, “*next*” point will be executed ending the *Iterative Leg*.

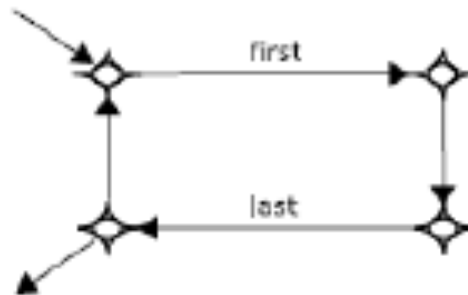


Figure 2.6 Iterative Leg

2.4.2.5. Legs not taken into account

I decided to not take into account Radius to fix leg (RF leg), Holding pattern (HF leg) and Eight leg since they are not useful maneuvers for my work and for a drone in general. All curved paths are avoided along this master thesis since a drone can use 90 degrees maneuvers to turn, to hold and to scan.

2.5. FPs Strategy

In order to build my own flight plans, I used the Neighborhood AirSim environment as reference to collect a list of coordinates and to plan a series of maneuvers. In chapter 3 the Neighborhood environment and its reference frame system as well as the conversion parameters from Geographical coordinates to AirSim NED coordinates will be deeply analyzed.

It has to be noticed that all tables and graphs, as well as in the text section, “*m*” stands for meters, “*o*” stands for degrees, “*s*” stands for seconds.

2.5.1. Single-Drone Flight Plan

After take off (NED coordinates: 0, 0, -2), *Carlo* enters the first stage consisting of two consequently *To Fix Leg* to arrive at the *Intersection Leg* point. The parameters of the two points for both FPs are summarized in the table below (Table 2-1).

STAGE 1				
Point Name	Speed [m/s]	North Coordinate [m] / Latitude [°]	East Coordinate [m] / Longitude [°]	Altitude [m] / Altitude [m]
zero-point (N0)	5	125 / 47.64260464285712	0 / -122.140365	-20 / 143.199297198
first-point (N1)	5	125 / 47.64260464285712	125 / -122.1386985308925	-40 / 163.199297198

Table 2-1 N0 and N1 waypoints coordinates

Image 2.7 shows an example of *To Fix Leg* (N0 Point, NED FP) written with XML language.

```

<leg id="zero-point" xsi_type="fp_TFLeg">
  <dest>
    <name>N0</name>
    <north_coordinate>125</north_coordinate>
    <east_coordinate>0</east_coordinate>
    <altitude>-20</altitude>
    <speed>5</speed>
    <next>first-point</next>
  </dest>
</leg>

```

Figure 2.7 Example of To Fix Leg

Entering the second stage which is composed only by the *Intersection Leg*, the user can choose among a list of three possibilities through an input given from keyboard. The following image (2.8) shows the heading of this leg and the “nextList” composed by the three possibilities, without his body.

```

<leg id="second-point" xsi_type="fp_IntersectionLeg">
  <nextList>third-point-a third-point-b third-point-c</nextList>
</leg>

```

Figure 2.8 Heading of Intersection Leg

The first possibility (third-point-a, NED FP) is a *Parametric Leg* with a trackseparation of 50m starting at point1 coordinates. The image 2.9 shows the XML code for this choice (NED FP).

```

<leg id="third-point-a" xsi_type="fp_Scan">
  <dest>
    <coordinates>0 0</coordinates>
  </dest>
  <trackseparation>50</trackseparation>
  <area>
    <point1>125 -125</point1>
    <point2>-125 -125</point2>
    <point3>-125 125</point3>
    <point4>125 125</point4>
  </area>
  <speed>5</speed>
  <altitude>-40</altitude>
</leg>

```

Figure 2.9 Example of Parametric Leg

The second one (third-point-b, Image 2.10, NED FP) is an *Iterative Leg* composed by a To Fix Leg followed by a scan with a trackseparation of 125m. The UpperBound value has been fixed to 2. After the repetition of this path, Carlo will directly fly to fourth-point with a *To Fix Leg*.

```

<leg id="third-point-b" xsi_type="fp_IterativeLeg">
  <next>fourth-point</next>
  <body>third-point-b-one third-point-b-two</body>
  <upperBound>2</upperBound>
  <first>third-point-b-one</first>
  <last>third-point-b-two</last>
</leg>
  <leg id="third-point-b-one" xsi_type="fp_TFLeg">
    <dest>
      <name>N3B1</name>
      <north_coordinate>-125</north_coordinate>
      <east_coordinate>125</east_coordinate>
      <altitude>-40</altitude>
      <speed>5</speed>
      <next>Home1</next>
    </dest>
  </leg>
  <leg id="third-point-b-two" xsi_type="fp_Scan">
    <dest>
      <coordinates>0 0</coordinates>
    </dest>
    <trackseparation>125</trackseparation>
    <area>
      <point1>-125 -125</point1>
      <point2>-125 -125</point2>
      <point3>125 -125</point3>
      <point4>125 125</point4>
    </area>
    <speed>5</speed>
    <altitude>-40</altitude>
  </leg>
  <leg id="fourth-point" xsi_type="fp_TFLeg">
    <dest>
      <name>N4</name>
      <north_coordinate>125</north_coordinate>
      <east_coordinate>125</east_coordinate>
      <altitude>-40</altitude>
      <speed>5</speed>
      <next>Home1</next>
    </dest>
  </leg>

```

Figure 2.10 Example of Iterative Leg

The last one (third-point-c) is another Scan starting in a different point and with a trackseparation of 62.5m.

Once the path related to the possibility chosen is finished, to come back to the starting point *Carlo* will enter the third stage composed by two consequently *To Fix Leg* (Table 2-2). Then, the drone will land exactly where it took off.

GO HOME STAGE				
Point Name	Speed [m/s]	North Coordinate [m] / Latitude [°]	East Coordinate [m] / Longitude [°]	Altitude [m] / Altitude [m]
Home1	4	60/ 47.642021060971416	0 / -122.140365	-20 / 143.199297198
Home2	2	0 / 47.64260464285712	0 / -122.140365	-4 / 127. 199297198

Table 2-2 Home1 and Home2 waypoints coordinates

Images of the complete XML file are in “Annexes” (Chapter 8).

2.5.2. Multiple-Drones Flight Plans

Paolo will be the first drone to start its mission; its FP consists of a *To Fix leg* and a successively *Iterative leg* composed by four *To Fix legs*. *Paolo* will fly from corner to corner, covering all the Neighborhood external perimeter for twenty times. After this *Iterative leg*, the drone will come back to the starting point. *Francesca* takes off with *Paolo*, but it has to wait to execute the mission until the user gives an input from keyboard. Its FP only consists of an *Intersection leg*: the user has to choose among four different scanning paths, one for each section of the Neighborhood environment corresponding to one of the four quadrants of the Cartesian Plane (more details in section 3.3). The idea is that *Paolo* starts to monitor the Neighborhood covering the external perimeter; if it finds something wrong in one of the section of the map, *Francesca* will start its scanning path over that area. Images of the complete XML files are in “Annexes” (Chapter 8).

2.6. XML Processing

In chapter 4, it will be explained how parse an XML file on Python and how reach the different childs through their tags and attributes. Moreover, the implementation of the Python functions representing the four different legs taken into account will be deeply described. Finally, both data-storage and transmission-commands codes will be outlined to show how the dynamic interfaces store waypoints informations and turns these data into commands for the drones in the simulator environment.

Chapter 3

AIRSIM SIMULATOR

3.1. Overview

In order to save money and time, simulators usefulness has been increasingly growing in the last years. Simulation provides necessary data and it allows users to test mathematical models as well as the behavior of facilities and dynamic systems. The great advantages are the cost and time optimization. Since a drone can crash numbers of time in the simulator, we can deeply learn how it behaves and how fix problems arising during tests.

This master thesis will take advantages of this technology making a step toward the implementation of an autonomous drone, which can move without any external aid. The dynamic interface will help ICARUS project (UPC) to test several flight plans on several AirSim environments without any kind of dependencies.

3.2. General Features: Unreal Engine and AirSim

“Unreal Engine is a complete suite of creation tools designed to meet ambitious artistic visions while being flexible enough to ensure success for teams of all sizes” [19]. In 1998 Epic Games presented “*Unreal Engine (UE)*” that is an engine developed for a wide variety of game genres, especially first-person shooters games. Later on, this exceptional software has found several other applications and actually, the most recent version is UE4 (released in 2014) [15].

AirSim (2017) is an open-source simulator still under development and it takes advantage of Unreal Engine and its environments. UE environments are shaped with physics and aerodynamics models, taking into account all the forces and torques acting on the vehicles. All these models are taken as inputs by the physics engine to allow the computation of the vehicle kinematics-state in the simulated world. AirSim is a very complex and well-made simulator and it is not easy to be used. Figure 3.1 shows the AirSim interface respectively with a car in the “City Virtual Environment” and a drone in the “Neighborhood Virtual Environment”.

AirSim has been provided with a set of APIs to control vehicles in the simulator through several programming languages. This cross-platform capability allows control transmission from both C++ and Python programming codes as well as a support for Windows and LinuxOS platforms.

Moreover, one of the most important AirSim capability concerns deep-learning as well as reinforcement-learning algorithms for vehicles moving independently [18].



Figure 3.1 Car in City Environment - Drone in Neighborhood Environment

3.3. Neighborhood Environment: Reference Frame System

The environment chosen for the implementation of the XMLs is the “*Neighborhood*” AirSim environment. All the coordinates that appear in the six Flight Plans have been rationally taken to allow Carlo, Paolo and Francesca safe flight.

AirSim reference-frame-system is a North, East, Down frame (NED). Hence, x-axis represents North-coordinates, y-axis represents East-coordinates and z-axis (representing altitude) is pointing down, so all the altitude coordinates will be negative.

Neighborhood environment (figure 3.2) can be approximated to a square with a side of $250m$. The default starting-point (*NED coordinates: $0m, 0m, \approx -2m$*) is at the diagonals intersection-point and the front camera is initially pointing towards the positive North direction. Since the z-axis is pointing down, taking advantages of the “right-hand rule” it will be clear that the positive East direction is on the drone right-side at the default starting point.



Figure 3.2 Neighborhood Reference Frame System

AirSim Simulator only takes NED coordinates as input hence, for the Geographical FP it is required that the interfaces convert the coordinates before executing commands. To find the conversion parameters (ΔLat and $\Delta Long$), the Python's library *geographiclib* has been used; the altitude does not require a conversion parameter. In the table below symbols and values for the three initial Geographical coordinates (default starting point; evaluated with "getMultirotorState" AirSim's function), the two conversion parameters for the chosen environment (calculated with *geographiclib*) and the initial NED altitude are summarized.

SYMBOLS AND VALUES		
Data	Symbol	Value
Initial Latitude	Lat_0	47.64148237°
Initial Longitude	$Long_0$	− 122.140364°
Initial Geographical Altitude	h_{geo_0}	125.1653061m
Initial NED Altitude	h_{NED_0}	−1.966008902 m
Latitude conversion Parameter	ΔLat	8.99415308151351 $\times 10^{-6} \text{ } ^\circ/m$
Longitude Conversion Parameter	$\Delta Long$	1.33084196690962 $\times 10^{-5} \text{ } ^\circ/m$

Table 3-1 Default Starting Point Coordinates - Conversion Parameters

Knowing the current latitude (Lat), longitude ($Long$) and Geographical altitude (h_{geo}), we can evaluate the actual NED coordinates (N_C, E_C, h_{NED}) with the following equations (1, 2 and 3):

$$N_C = (Lat - Lat_0)/\Delta Lat \quad (1)$$

$$E_C = (Long - Long_0)/\Delta Long \quad (2)$$

$$h_{NED} = h_{NED_0} - (h_{geo} - h_{geo_0}) \quad (3)$$

Moreover, it is possible to change the default strating point coordinates. Figure 3.3 shows the "setting.json" file and how to set these coordinates.

```
{
  "SeeDocsAt": "https://github.com/Microsoft/AirSim/blob/master/docs/settings.md",
  "SettingsVersion": 1.2,
  "SimMode": "Multirotor",
  "OriginGeopoint": {
    "Latitude": 47.641468,
    "Longitude": -122.140165,
    "Altitude": 122
  }
}
```

Figure 3.3 Set default starting point coordinates in "Setting.json"

3.4. Surveillance Mission

For the development of the project, according to the surveillance-mission objective and its scan-paths, I chose a service altitude of $-40m$ in order to avoid any conflict with trees and houses. The real drone will use the bottom camera to scan and control the area and it will transmit real-time photos from above, so the AirSim frontal camera has not been employed (neither to detect and avoid obstacles). Moreover, to avoid conflicts between drones in the multiple-drones scenario, the first drone will fly at $-30m$ and the second one at $-40m$. Since the first drone follows only the external perimeter, it will not have any collision with trees or houses. In the “Recommendations” section (6.3) all the responsibilities of the FPs designer will be explained.

In chapter 4, the connection-to-AirSim code of the interfaces and how transmit commands to the drones in the AirSim environment taking advantages of Python and its APIs will be explained.

Chapter 4

DEVELOPMENT OF THE FP INTERPRETER

4.1. Python features

Python (see [8]) is an interpreted, interactive and object-oriented programming language. “Interpreted” means that Python directly launches the source file. “Interactive” means that it is allowed the handwriting of instructions directly in the command prompt. Nevertheless, it is possible to download several *Integrated Development Environments (IDE)* that simplify Python usage.

This programming language takes advantages of moduls (import command), exceptions and their management (try, except, finally, else instructions), dynamic typing and high-level and high-class data like lists, set and dictionaries and their *comprehension*. These and others are the features that make Python one of the best programming language in the world: e.g. strings, tuple, mandatory indentation, standard libraries, slicing, application libraries, functions, dictionaries etc. The syntax is extremely clear and easy; the usage of Boolean values (True, False, None) is allowed.

4.2. A little bit of history

Python was created by Guido Van Rossum and released in 1991 for free directly on the web. Guido is a very famous expert in programming languages and, immediately after its released, Python gained popularity among the informatic community. Guido Van Rossum gave this name to its “creature” in honor of the 70’ rock group *Monty Python*, who choose this name because “*it sounded funny*”.

4.3. Single-Drone Interface

Each sub-section presents one of the several parts that make up the code. Moreover, this section has been divided in as many sub-sections as the number of topics covered.

4.3.1. AirSim

AirSim’s APIs for Python allow us to connect and disconnect our script from the AirSim simulator; moreover, take off and landing are directly executed by two different functions.

Figure 4.1 (from code line 612 to 633) shows how to connect the script to the simulator, enable APIs commands, get the drone state and store the initial coordinates into variables. Initial coordinates are necessary for plots, for landing and for the coordinates conversion if the Geographical FP is used. Figure 4.2 (code lines 830 and 831) presents the code lines to execute the take off. The method Async calls future, so we have to wait until the take off is completed. To do this, we use the function “sleep” of the “time” library; the input of this function represents the number of seconds to wait.

```

client = airsim.MultirotorClient()
client.confirmConnection()
client.enableApiControl(True)
client.armDisarm(True)

state = client.getMultirotorState()
s = pprint.pformat(state)
print("state: %s" % s)

x_home = state.kinematics_estimated.position.x_val
y_home = state.kinematics_estimated.position.y_val
z_home = state.kinematics_estimated.position.z_val

lat0 = state.gps_location.latitude
long0 = state.gps_location.longitude
h_geo0 = state.gps_location.altitude

arcWE = geodesic.Geodesic.WGS84.Inverse(lat0, long0-0.5, lat0, long0+0.5)
arcNS = geodesic.Geodesic.WGS84.Inverse(lat0-0.5, long0, lat0+0.5, long0)

lat_conv = 1 / arcNS['s12']
long_conv = 1 / arcWE['s12']

```

Figure 4.1 Connection to the simulator code lines

```

client.takeoffAsync()
time.sleep(4)

```

Figure 4.2 Take off code lines

Figure 4.3 (from code line 856 to 859) shows how to implement drone landing and disconnect the script from the simulator. Since the last point of the FP is perfectly above the landing point (default starting point), instead of using the “land” function, “moveByVelocityZAsync()” function has been used to have a vertical movement to reach the ground (“join()” replaces “time.sleep” function).

```

client.moveByVelocityZAsync(0, 0, z_home, 2, airsim.DrivetrainType.MaxDegreeOfFreedom, airsim.YawMode(False, 0)).join()
client.armDisarm(False)
client.enableApiControl(False)

```

Figure 4.3 Landing code lines

To allow *Carlo*’s movement, the remaining function used in the code is the “moveToPositionAsync()”. This function takes at least four inputs: north coordinate, east coordinate and altitude of the point to be reached and the cruise speed. In all the Python’s functions only this AirSim’s function is used to move the drone and to execute all the possible maneuvers. Again, “Async” method calls future, so “time.sleep” function is required. In sub-section 4.3.9.2 it will be shown how wait the right amount of time to perfectly complete each section of the FP with my “check_position” function.

4.3.2. XML Parsing

In order to make Python capable of reading and decoding an XML file, this file has to be parsed. To do this, it can be used one of the many Python’s libraries that allow the processing of XML files.

The “ElementTree” (ET) library (the import of this library will be shown in sub-section 4.3.8) has been chosen. Using the function “ET.parse()” that takes a string containing the name of the file as input, the XML FP can be parsed and saved into a variable “tree”. It has to be noticed that the file has to be in the same folder of the main script. After that, to decode the file, other two lines of code have to be added.

Figure 4.4 (from code line 588 to 596) shows the implementation of the parsing code for the XML FPs. As the code starts to run, the user has to choose which flight plan wants to parse and use it to move *Carlo* in the simulator environment. The variable “FP” will be set giving an input from keyboard. If “FP” is set to “0”, the parsed geographical flight plan will be used; if “FP” is set to “1”, parsed NED flight plan will be used. The last two lines of code allow us to decode into strings the whole XML FP.

This part of code could be improved by browsing the desired file to be parsed in the machine. It was not a key point of the project but, with an eye towards the marketing of the interface, this improvement can be easily done.

```
FP = input("Which Flight Plan do you want to use? Please, select 0 for Lat/Long FP or 1 for NED FP:")
if FP == '0':
    tree = ET.parse('DroneFlightPlan_Neighborhood_Final_Version_LongLat.xml')
elif FP == '1':
    tree = ET.parse('DroneFlightPlan_Neighborhood_Final_Version_meters.xml')
root = tree.getroot()
ET.tostring(root, encoding='utf8').decode('utf8')
```

Figure 4.4 XML Parsing code lines

4.3.3. Flight Plan Processing

From code line 637 to 826, it has been implemented a *dictionary* (*waypoints_data*) aimed at storing all the informations contained in the Flight Plan.

First of all, an empty dictionary to be filled has to be created. After that, the variable “FP” (previous sub-section) will be the discriminating factor in the selection of the right “if-loop-branch”. Indeed, the code lines to search inside the geographical flight plan are different with respect to the code lines to search inside the NED one. It has to be noticed that the working process is the same but the content to be searched inside the decoded XML is different (i.e. latitude or n_coord, longitude or e_coord, ...). Moreover, the other difference between the two “if-loop-brunches” is the presence of the code lines aimed at converting the geographical coordinates of the geographical flight plan (“FP=0”) into NED coordinates for AirSim. Hence, the dictionary will only contain NED coordinates (whatever XML has been used).

The working process to search the required informations inside the XML file is very cumbersome. With a “for loop”, we can iterate over the whole XML (variable “tree”) to search all the objects with a “tag = leg”. The name of each leg is stored as element of the dictionary and it will in turn be a dictionary (nested dictionary). Then, depending on the leg’s attribute that specifies the leg type, all the significant parameters will be saved inside this nested dictionary. In the end, a dictionary containing as many elements as the flight-plan-waypoints are (an unique key-word identifies each element) is obtained; these elements are themselves dictionaries containing as many elements

as the significant parameters of each leg are. Each significant parameter will be saved with its key-word. For a *To Fix Leg* the significant parameters are 4 (x-coord, y-coord, altitude and speed); for a *Scan Leg* are 6 (trackseparation, x-coord and y-coord of the “point1”, altitude, speed and the number of tracks); for an *Iterative Leg* are 3 (name of the points in the “body”, name of the “next point” and the “UpperBound” value); for an *Intersection Leg* are 2 (name and amount of the points contained in the “nextList”).

This dictionary allow us to store these parameters that will be needed to perform the required maneuvers. Each leg function contains code lines to search inside the dictionary (using the right key-word). Then, storing the elements into variables, the function will be able to execute the desired path.

4.3.4. Processing of Flight Plan Legs

To make the drone capable of following the required path, I implemented four functions, one for each type of leg taken into account along this master thesis. It has to be noticed that the “check position” function is employed everytime an AirSim’s movement function is used inside these “legs functions”. Its utility will be clarified in sub-section 4.3.9.2. Moreover, assuming that we want to plot graphs at the end of the simulation (LOG = “ON”, 4.3.6), a certain number of code lines are implemented to store time and position data.

4.3.4.1. Tf_leg Function

The first function to be presented is the *To Fix Leg* function (code lines 75 and 76). Image 4.5 shows that this function only consists of a “moveToPositionAsync” AirSim’s function. This calls will return a straight movement from the current position to the point described by the four inputs passed to this function (North coordinate = n_coord, East coordinate = e_coord, altitude = alt, velocity = speed).

```
def tf_leg(n_coord, e_coord, alt, speed):
    return client.moveToPositionAsync(n_coord, e_coord, alt, speed)
```

Figure 4.5 To Fix Leg Processing code lines

4.3.4.2. Scan Function

The scan function (from code line 81 to 391) allows drones to perform a scan path over the desired area. It takes as input the coordinates of the scan starting-point (n_coord, e_coord, h, v) and the trackseparation (ts). These inputs represent the size of the scan area. The last input (i) is obtained by dividing the side length of the scan-area by the trackseparation; this number represents the number of tracks of the scan.

First of all, a *To Fix Leg* will bring the drone at the scan starting-point. After that, the waypoints forming the scan path will be evaluated, depending on the side length of the scan-area and the trackseparation as well as the starting point coordinates. Once the points are calculated, a “for loop” containing a “moveToPositionAsync” AirSim’s calls will pass point by point all the waypoints previously stored in a list.

This function is composed by an external “if loop” (one “if” and three “elif”, one for each corner of the scan-area, including the possibility that the north coordinate or the east coordinate of the corner can be equal to zero). This is due to the different behaviour of

the scan path which depends on the starting point coordinates. Then, inside each branch of this “if loop”, there is another “if loop” (one “if” and one “elif”). This is due to the number of tracks (i) that can be odd or even.

4.3.4.3. Iterative_leg Function

At the beginning of this function (from code line 396 to 446), the whole path (composed by one or more legs) to be iterated is stored. Then, using a “while loop” it can be ensured that the drone will perform the desired path a number of time equal to the “UpperBound” limit. Hence, the code will run inside this loop as long as a counter (starting from zero and updated at the end of each iteration) will be minor than the UpperBound value. After that, the “next point” coordinates are evaluated and passed to the *To Fix Leg* function.

The single-drone interface allows us to put another Iterative Leg or an Intersection Leg as leg of the path to be iterated (as well as to-fix legs and scan legs).

4.3.4.4. Intersection_leg Function

This function takes as input the option corresponding to the user selection and the name of the point to be reached. Then, the respective leg function will be used depending on the leg type.

The single-drone interface allows us to put an Iterative Leg or another Intersection Leg as possible choice to be selected (as well as to-fix legs and scan legs).

4.3.5. Flight Recording

In order to plot the path of the drone in the AirSim environment (LOG = “ON”, 4.3.6), the position of the drone while it is moving around the map has to be stored. I chose to create an empty spread sheet that is filled by “get_data()” function (sub-section 4.3.9.1).

Figure 4.6 (from code line 602 to 606) shows how create an spread sheet, how add a worksheet and how write some text into a cell. “worksheet.write()” take the row’s number as first input, the column’s number as second input and a string containing the text to be written as third input.

Figure 4.7 (code line 861) represents the closing function for our spread sheet. After the spread sheet has been closed, it can be found in the same folder of the main script.

```
data_collection = xlswriter.Workbook('Data.xlsx')
worksheet = data_collection.add_worksheet()
worksheet.write(0, 0, "North Coordinates")
worksheet.write(1, 0, "East Coordinates")
worksheet.write(2, 0, "Altitudes")
```

Figure 4.6 Spread Sheet Creation code lines

```
data_collection.close()
```

Figure 4.7 Spread Sheet Closure code line

4.3.6. Visualization of Flight Recording

At the beginning of the main program, once the XML file is parsed, the user has to choose if he wants to plot graphs at the end of the execution of the FP or not. As previously said, this option will affect the `check_position` function. The LOG variable will be stored through an input from keyboard (code line 598). To plot the graphs, "ON" has to be written; on the contrary, to avoid graphs, "OFF" has to be written.

```
LOG = input("To run the code with plots enter ON; to run the code without plots enter OFF: ")
```

Figure 4.8 LOG variable code line

After the disconnection from AirSim and the spread sheet closure, the code presents all the lines to plot (from code line 867 to 1044) the desired significant graphs. These graphs will only appear the LOG variable is set to "ON". I chose to plot 6 graphs for each choice of the *Intersection leg* that will be shown in Chapter 5.

4.3.7. Execution of the FP

From code line 837 to 852 the creation of the empty vectors that will contain all the data to be plotted can be found; moreover, few code lines to execute the XML FP can be also found (we need only to call the "stage" function inside a for-loop to pass every stage name).

4.3.8. Libraries Used

At the beginning of the script, all the libraries imports required to execute the code (Figure 4.9, from code line 5 to 20) can be found. The following list wants to give a quick look at all libraries purposes:

- "airsim" library allow us to interact with the simulator;
- "pprint" library is used to print Carlo's state parameters after the connection to the simulator;
- "time" library allow us to evaluate the time required by the drone to execute each stage; moreover, this library is necessary to allow Python to sleep while the drone is reaching the desired position ("check_position" function);
- "math" library contains all the mathematical operations as the square root ("sqrt" function) and others;
- "xlsxwriter" library allow us to write into the excel file;
- "openpyxl" provides the functions needed to open the excel file once it is closed as well as to use all the data inside to plot graphs;
- "xml.etree.ElementTree" is the library that permits the parsing and the decoding of XML files;
- "geographiclib" library is used to find the conversion parameters to obtain NED Coordinates from Geographic Coordinates. This library allows the interface to perfectly work all around the simulated world of Unreal Engine;
- the remaining three libraries (mplot3d, numpy and matplotlib.pyplot) allow us to plot different 3D plots.

```
import airsim

import pprint
import time
from math import *

import xlswriter

import xml.etree.ElementTree as ET

from mpl_toolkits import mplot3d
import numpy as np
import matplotlib.pyplot as plt
import openpyxl

from geographiclib import geodesic
```

Figure 4.9 Libraries Used code lines

4.3.9. Other Functions

4.3.9.1. Get_data Function

“get_data” function (from code line 27 to 35) allows us to store the drone current position (NED coordinates) directly in an excel file.

This function is employed inside the “check_position” function (sub-section 4.3.9.2) and will be only used if the variable “LOG” is set to “ON” (sub-section 4.3.6).

Figure 4.10 shows the body of the function. The variable “column” is created in the main program (code line 845) and it is initially set to “0”. The first time that “get_data” is called, “column” will be set to “1” and it will be the counter that slides column by column inside the excel file. In the end an excel file composed by several columns containing all the positions covered by the drone is obtained, from take off to landing.

```
def get_data(state):
    global column
    column += 1
    n_c = state.kinematics_estimated.position.x_val
    e_c = state.kinematics_estimated.position.y_val
    h = state.kinematics_estimated.position.z_val
    worksheet.write(0, column, n_c)
    worksheet.write(1, column, e_c)
    worksheet.write(2, column, h)
```

Figure 4.10 Get_data Function code lines

4.3.9.2. Check_position Function

“check_position” function (from code line 40 to 70) can be considered the most important one of the code. To allow Carlo’s movement, the “moveToPositionAsync()” function has been used. “Async” method calls future, so “time.sleep” function is required.

“check_position” firstly evaluates the theoretical time required to perform the desired stretch of path (more details in section 5.7). After that, if the variable “LOG” is set to “ON”, it will be used a “busy-waiting-method”: python will sleep 0.1 seconds at a time *while* the desired position is not reached. Moreover, each 0.1 seconds “get_data” function will be called to store all the current positions covered by the drone. On the contrary, if “LOG” is set to “OFF”, we will not need plots and the function “get_data”: python will sleep the whole theoretical time at once. Then, a safety “while loop” will check if the desired position is reached or not; python will sleep 0.1 seconds at a time while checking. The second method presents some advantages, first and foremost the reduction of the machine workload.

4.3.9.3. XML Tree Traversing

“stage” function (from code line 573 to 583) is employed in the execution of the main program inside a “for loop”. This function takes as input the name of the stage to be performed, employs “get_first_child” function to create a list of all the stage waypoints and uses “leg” function inside a “for loop” to pass this list point by point. Moreover, it is evaluated the real-simulation-time to perform the whole stage that is stored in a vector. This vector will be used by the plot code-lines.

“get_first_child” function (from code line 522 to 568) takes as input the name of the stage to be performed and returns the stage’s first-children. This is a safety function aimed to perform the FP in the correct order and avoid the repetition of already covered waypoints.

“leg” function (from code line 487 to 517) takes as input the name of the point to be reached and returns one of the leg functions previously described (depending on the leg type corresponding to the point).

4.4. Multiple-Drones Interface

This section wants to show the tool used to allow Python’s simultaneous handling of two or more drones (threading). Moreover, the whole code will be presented and analyzed to underline the main parts that make it up.

4.4.1. Threading

To allow Python’s simultaneous handling of two or more drones, threading is required. “A thread is a separate flow of execution” (see [7]); the script will have two or more tasks to be simultaneously accomplished. Moreover, to obtain a perfect thread’s management, threads’ synchronization is required. A “ThreadPoolExecutor” has been used to create and start threads (it submits one function to each thread); a Python’s “Class” has been implemented to obtain threads’ synchronization through the “Lock” function of the “threading library”.

4.4.2. Libraries Used

At the beginning of the script, all the libraries imports required to execute the code (Figure 4.11, from code line 5 to 17) can be found. With respect to the sub-section 4.3.8, all the libraries to plot graphs are not required and two more libraries have been added.

“Concurrent.futures” library allows the creation and the launch of the required number of threads using a “ThreadPoolExecutor” and a “for-loop”; each thread (executor) will execute a specific function.

“Threading” library, in particular the function “Lock” of this library, allows a basic synchronization of the threads created by the “ThreadPoolExecutor”.

```
import airsim

import threading
import time
import concurrent.futures

from math import *

import xml.etree.ElementTree as ET

import numpy as np

from geographiclib import geodesic
```

Figure 4.11 Libraries used code lines

4.4.3. AirSim

After the import of the libraries, the code lines required for the connection to AirSim can be found. Then, the user has to chose how many drones will fly in the simulator environment though an input given by the keyboard (“population” variable); a “for-loop” will create the desired number of drones (each one with an unique name: Drone1, Drone2,...). Moreover, “population” variable is also used in the “for-loop” of the “ThreadPoolExecutor”.

To allow the take off of all the drones and to put them at different heights, a “for-loop” has been implemented using the “moveToZAsync” function. The first drone (Drone1) will hover at $-2m$ until all drones finish the take off; the second one (Drone2) will hover 3 meters above ($-5m$), the third one 6 meters above ($-8m$) and so on.

To implement the landing part, a specific function has been created (sub-section 4.4.4)

4.4.4. Execution of the FP

First of all, a Python’s “Class” (MainProgram) has been created (from code line 43 to 2058). Moreover, it has been initialized with the definition of the “self._lock” variable

that takes advantages of the “Lock” function of the “threading” library. To synchronize threads, “self._lock.acquire” and “self._lock.release” functions will be used.

The first function encountered in the class is the “execution” function (from code line 47 to 2042). It takes as input the “self” variable and the “drone number”; this second input identifies one of the multiple drones created at the beginning of the main script with a unique name. This function combines all the functions previously described for the single-drone interface. As a result, each thread will only use one function to execute all the FP. This approach imposes new limits on the interface that will be discussed in the “Recommendations for Multiple-Drones Interface” sub-section (6.3.3). Moreover, it has to be noticed that “self._lock.acquire” and “self._lock.release” functions are only used to parse the XML and to create the dictionary (from code line 49 to 275). After that, threads’ synchronization is obtained and the usage of these two functions is no longer required.

The second and final function belonging to the “MainProgram” class is the “landing” function. It takes as input the “self” variable and the “drone number”; this second input identifies one of the multiple drones created at the beginning of the main script with a unique name. Here, threads’ synchronization is obtained using “with self._lock:” code line (2046): each thread has to wait for the completion of the landing stage of the previous one.

Using a “for-loop”, the ThreadPoolExecutor will submit to each thread one of the functions contained in the MainProgram Class at a time. Then, to submit another function to each thread with another “for-loop”, all threads have to finish their tasks. From code line 2062 to 2072, the two “for-loops” required to submit the “execution” function and the “landing” function to the two threads can be found.

Chapter 5

PLOTS AND RESULTS

This chapter shows all the plots and results obtained running the single-drone dynamic interface. I tested both Geographical and NED flight plans; I run each FP three times, once for each *Intersection leg* possibility. I Decided to plot six significant graphs. The first one compares the theoretical path (waypoints of the FP) and the simulated path (the path followed by the drone in the simulator). The second plot compares the theoretical North coordinate with the simulated one (first graph), the theoretical East coordinate with the simulated one (second graph) and the theoretical Altitude with the simulated one (third graph). Finally, the last four graphs compare the theoretical and the simulated time spent to execute each stage of the FP and the total time spent for the mission. Hence, each section presents six plots and a table containing the detailed time values.

5.1. Plots and Data First Choice Intersection NED FP

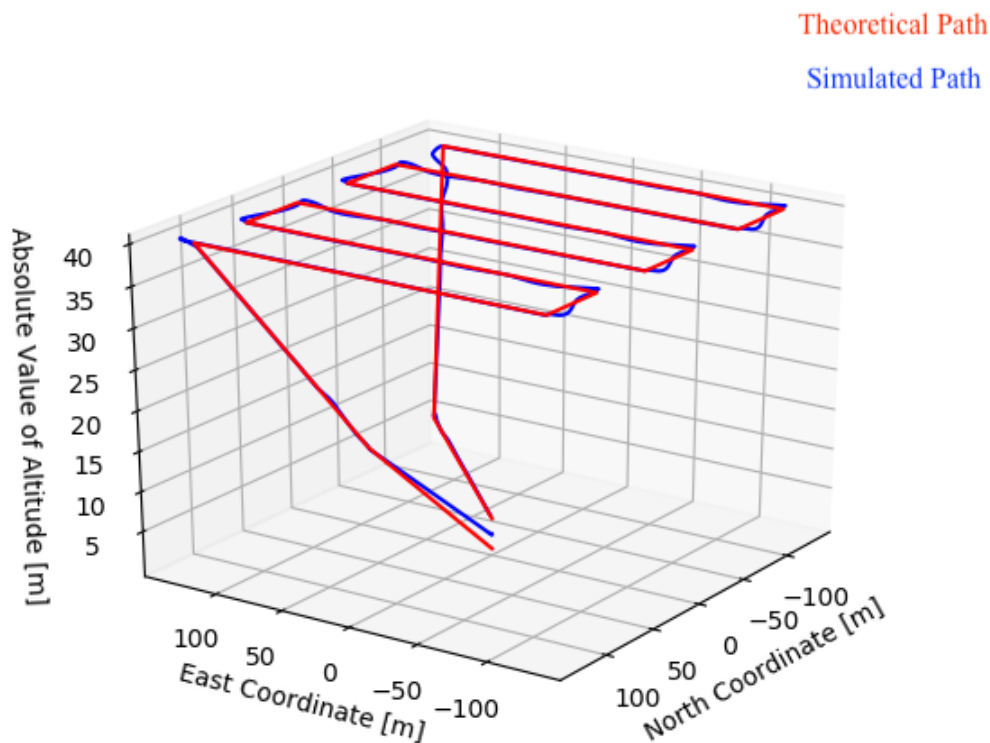


Figure 5.1 Theoretical Path - Simulated Path comparison

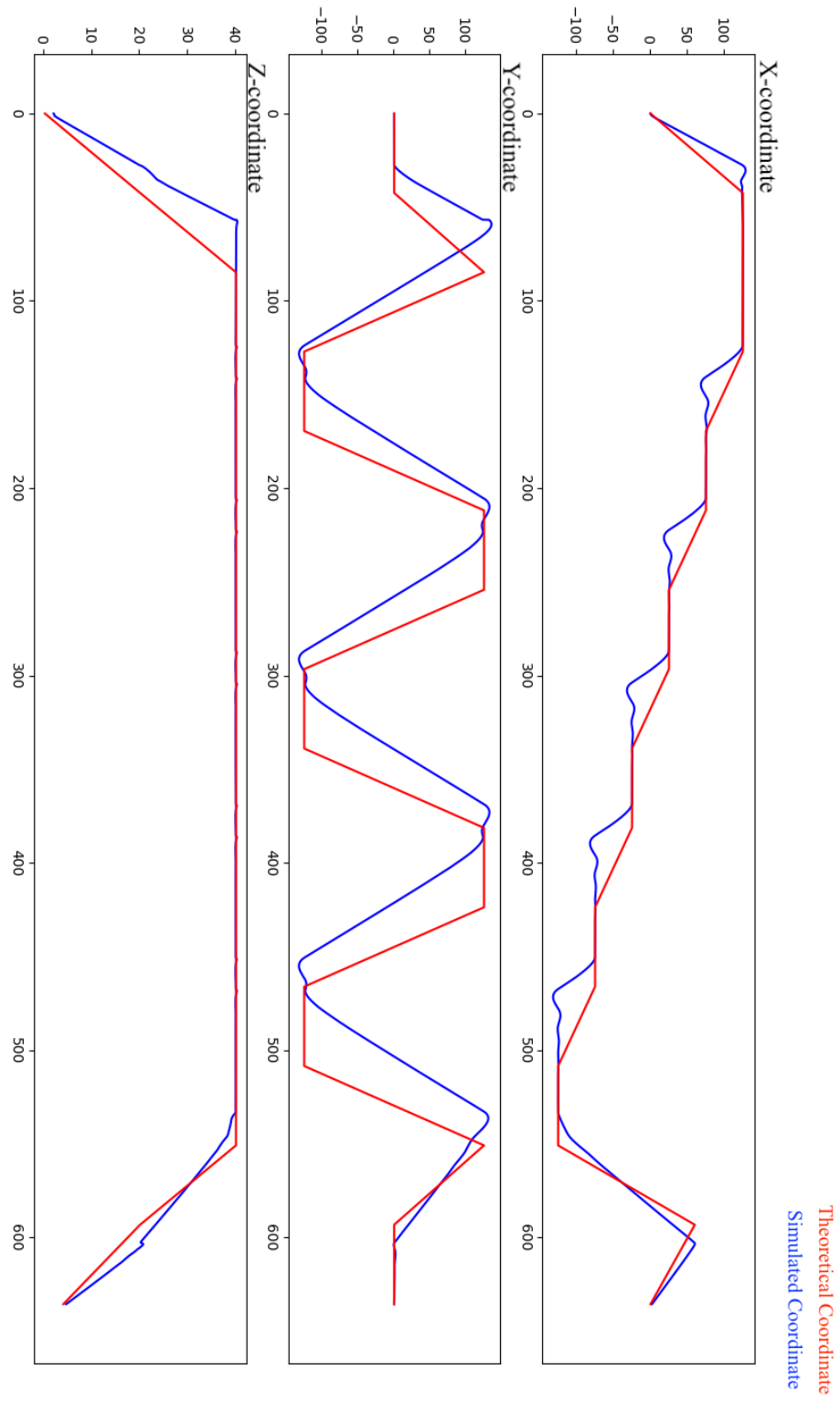


Figure 5.2 Theoretical Coordinates - Simulated Coordinates comparison

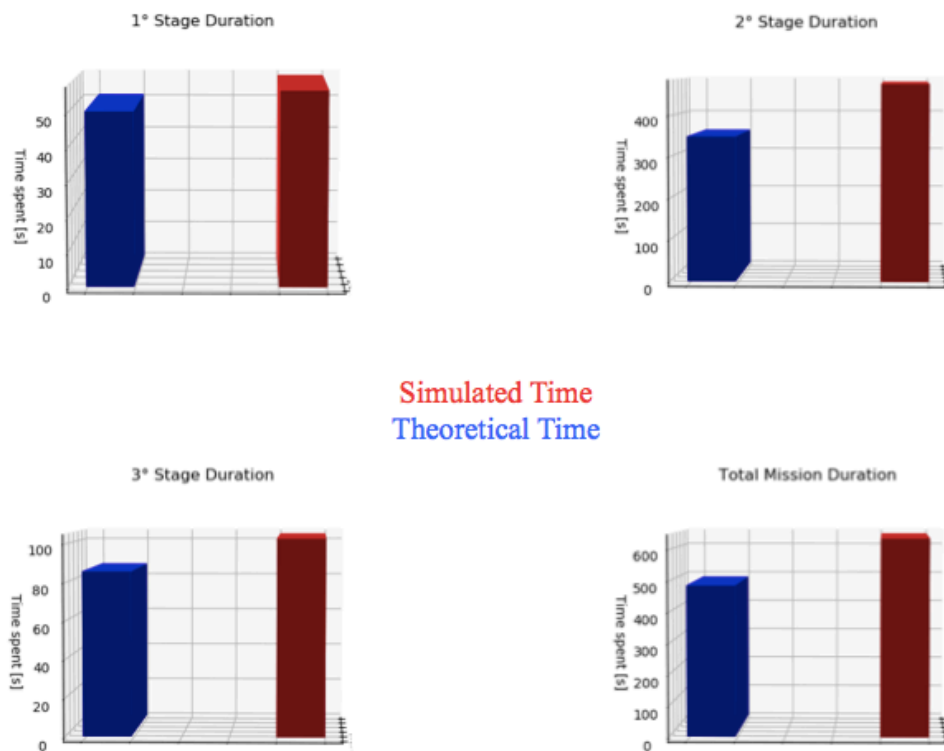


Figure 5.3 Theoretical Time - Simulated Time comparison

TIME TABLE		
	Theoretical Time [s]	Real Time [s]
1° Stage	50.59	56.45
2° Stage	349.01	474.05
3° Stage	85.95	102.88
Mission Duration	485.55	633.38

Table 5-1 Theoretical Time Values - Simulated Time Values comparison

The difference between theoretical and simulated time of the global mission is 147.83s, the 30,4% more than planned duration. In sub-section 5.7 it will be given a reasonable justification for this result.

5.2. Plots and Data Second Choice Intersection NED FP

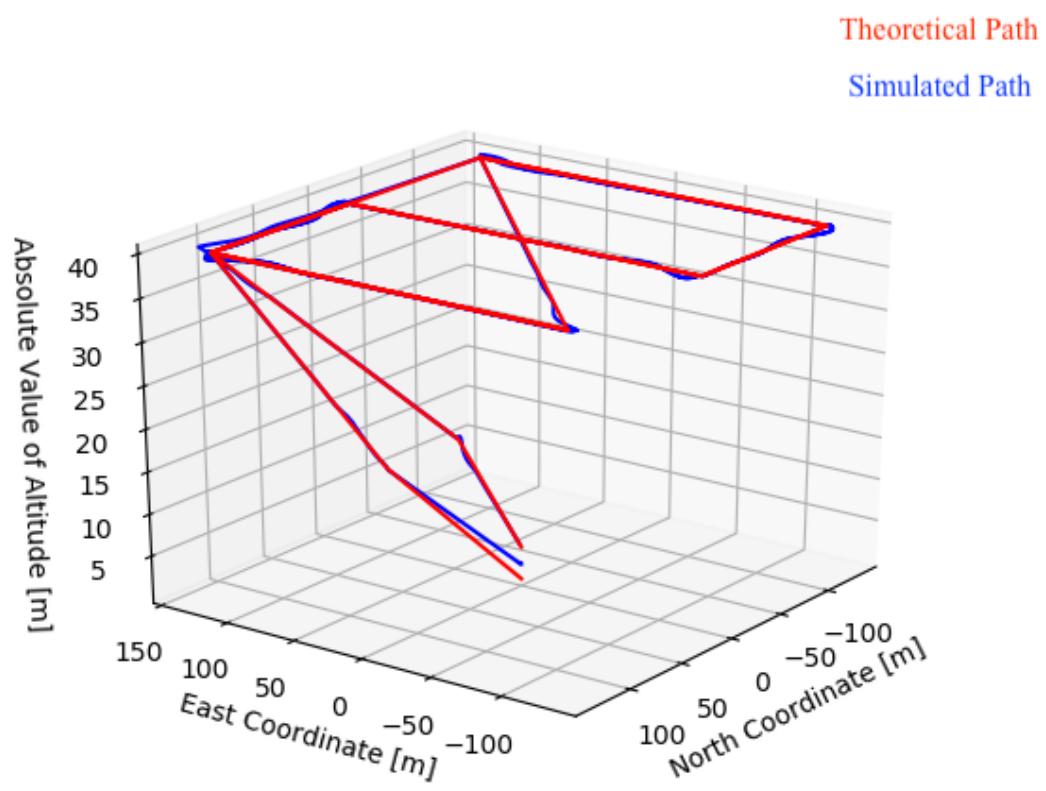


Figure 5.4 Theoretical Path - Simulated Path comparison

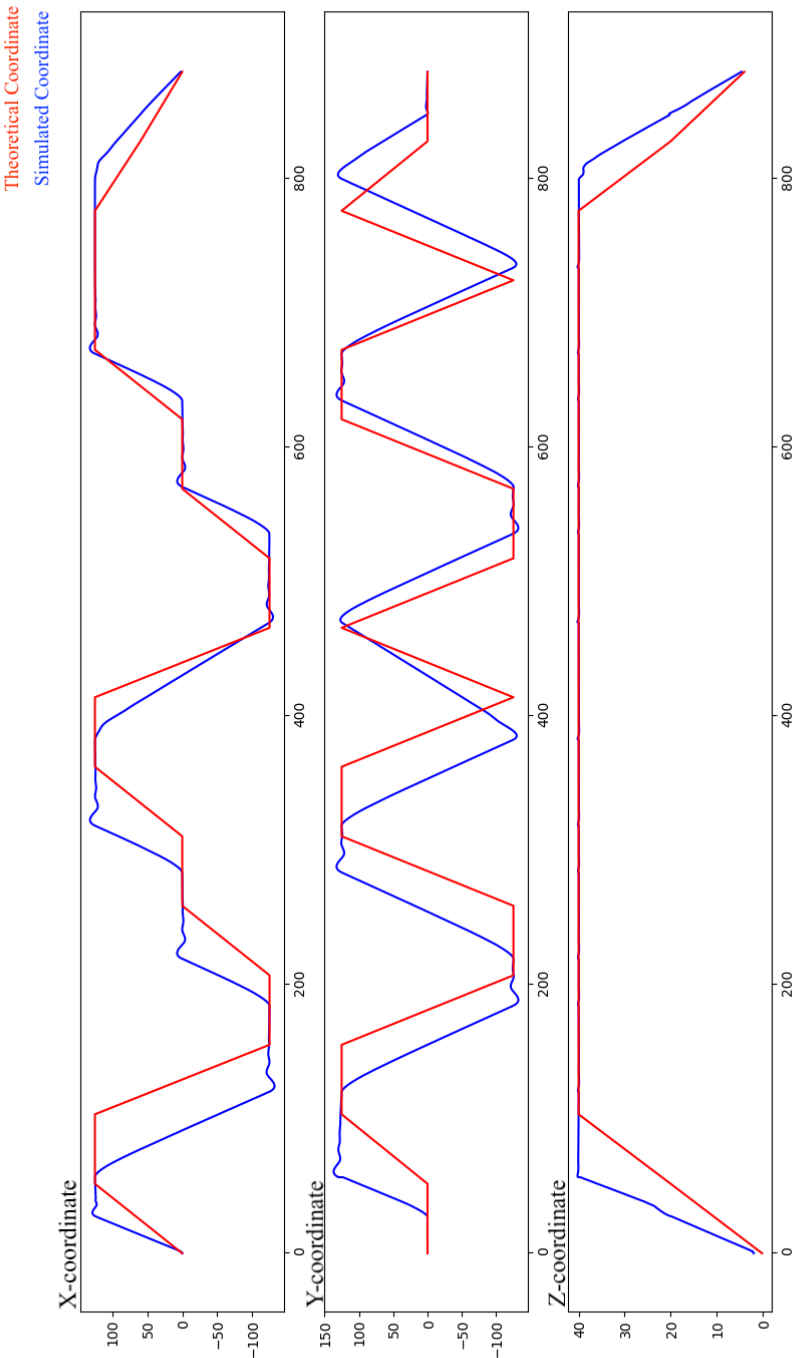


Figure 5.5 Theoretical Coordinates - Simulated Coordinates comparison

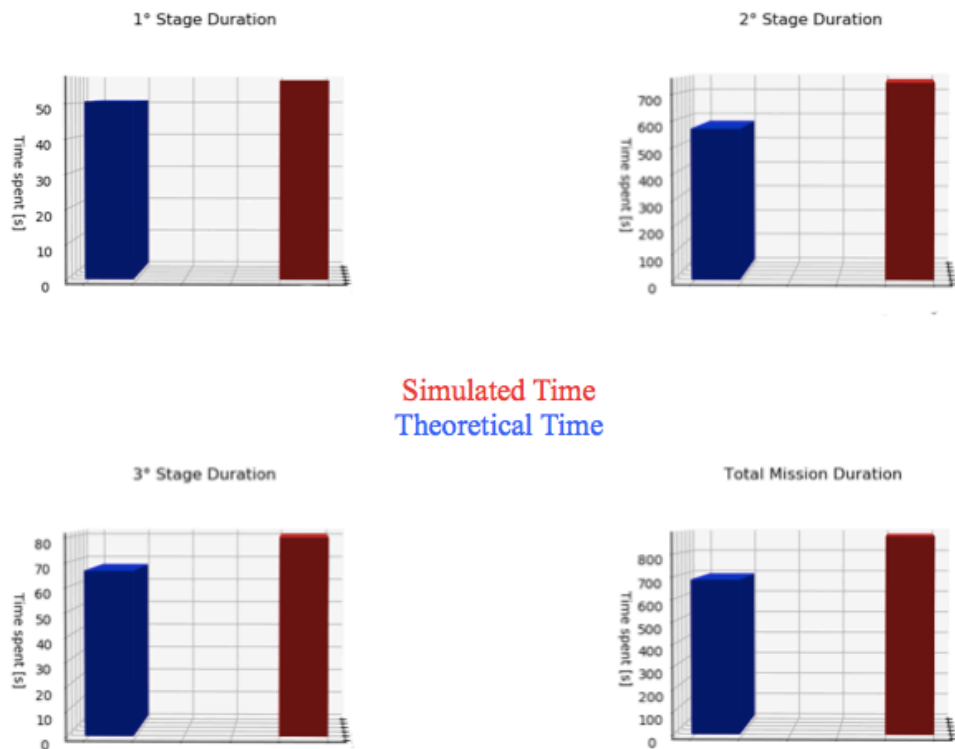


Figure 5.6 Theoretical Time - Simulated Time comparison

TIME TABLE		
	Theoretical Time [s]	Real Time [s]
1° Stage	50.59	56.58
2° Stage	570.05	742.62
3° Stage	66.78	80.16
Mission Duration	687.42	879.36

Table 5-2 Theoretical Time Values - Simulated Time Values comparison

The difference between theoretical and simulated time of the global mission is 191.94s, the 27.9% more than planned duration. In sub-section 5.7 it will be given a reasonable justification for this result.

5.3. Plots and Data Third Choice Intersection NED FP

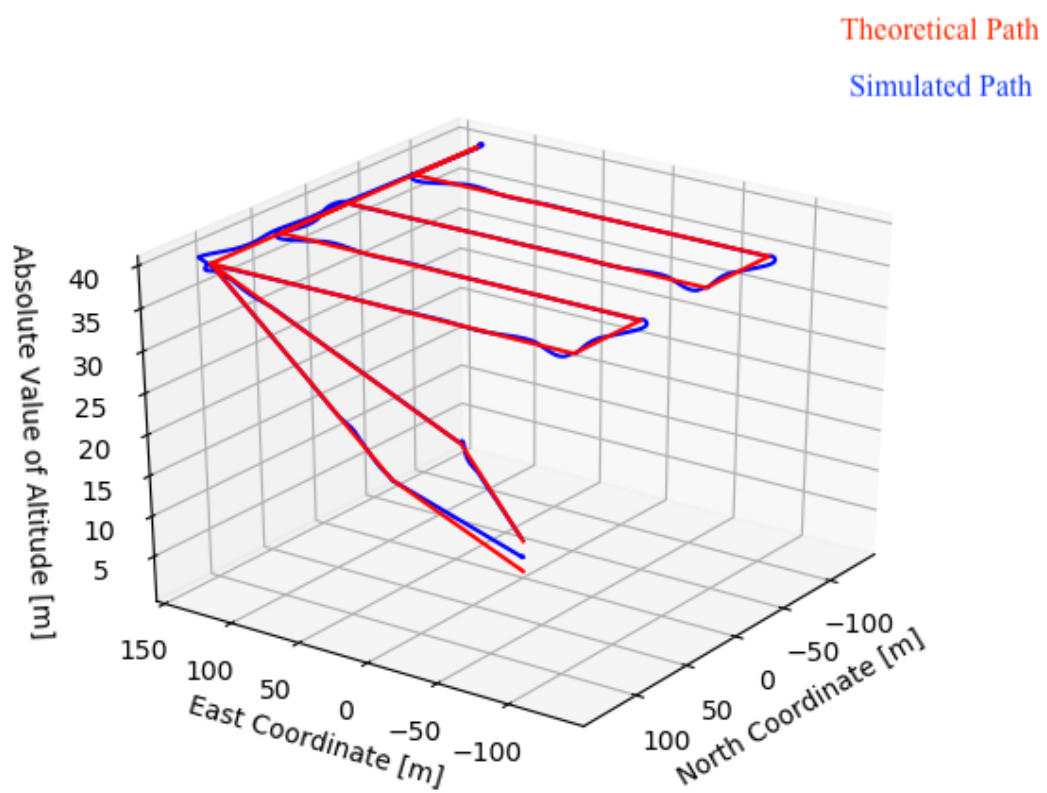


Figure 5.7 Theoretical Path - Simulated Path comparison

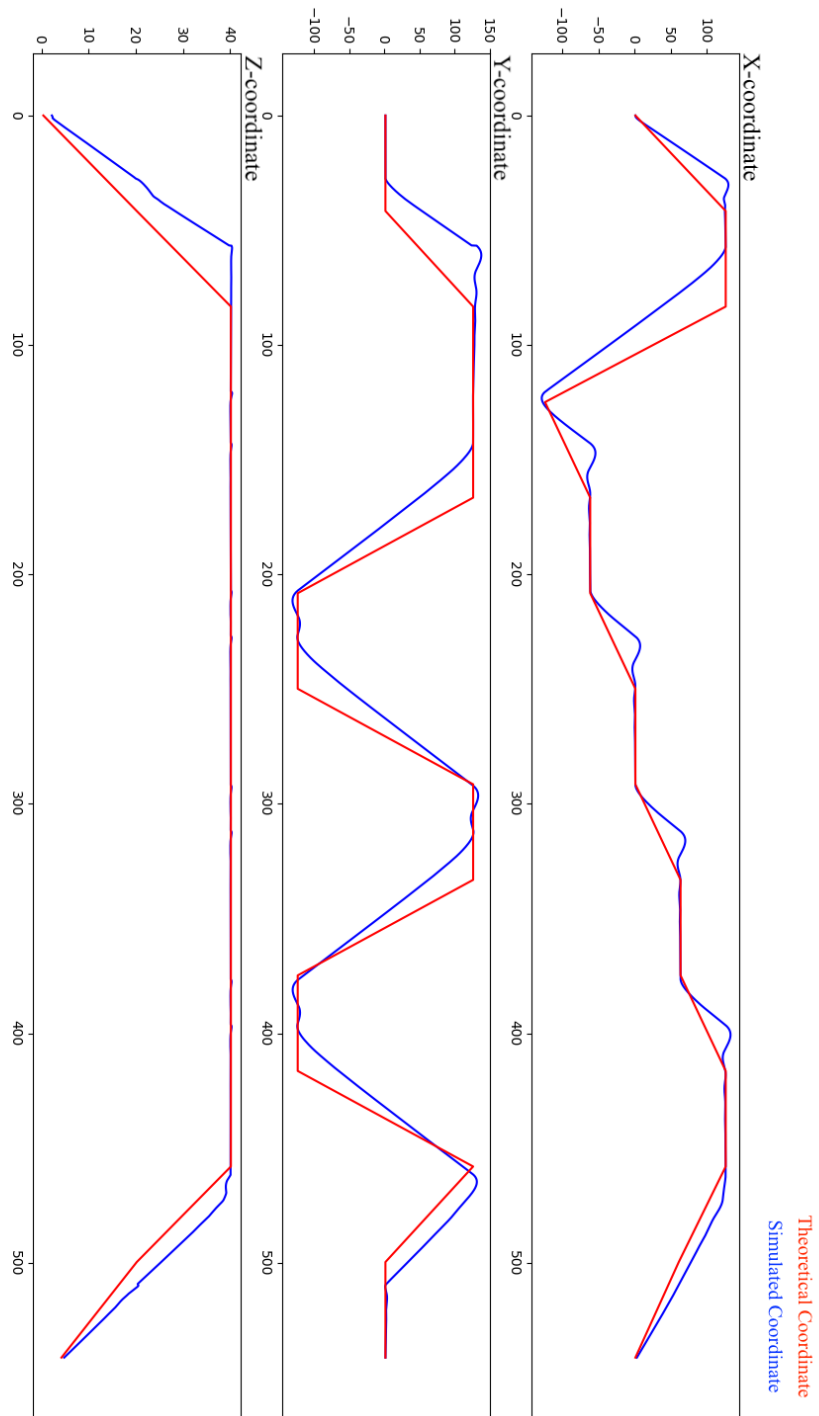


Figure 5.8 Theoretical Coordinates - Simulated Coordinates comparison

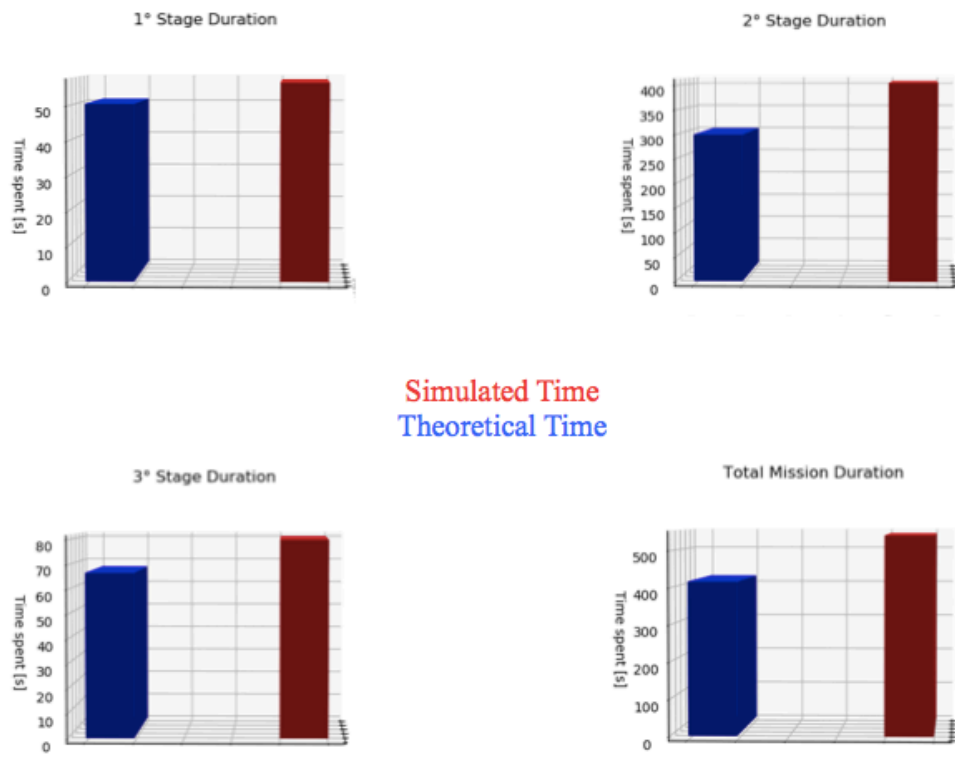


Figure 5.9 Theoretical Time - Simulated Time comparison

TIME TABLE		
	Theoretical Time [s]	Real Time [s]
1° Stage	50.59	56.44
2° Stage	299.85	404.19
3° Stage	66.71	80.05
Mission Duration	417.15	540.68

Table 5-3 Theoretical Time Values - Simulated Time Values comparison

The difference between theoretical and simulated time of the global mission is 123.53s, the 29,6% more than planned duration. In sub-section 5.7 it will be given a reasonable justification for this result.

5.4. Plots and Data First Choice Intersection Geographical FP

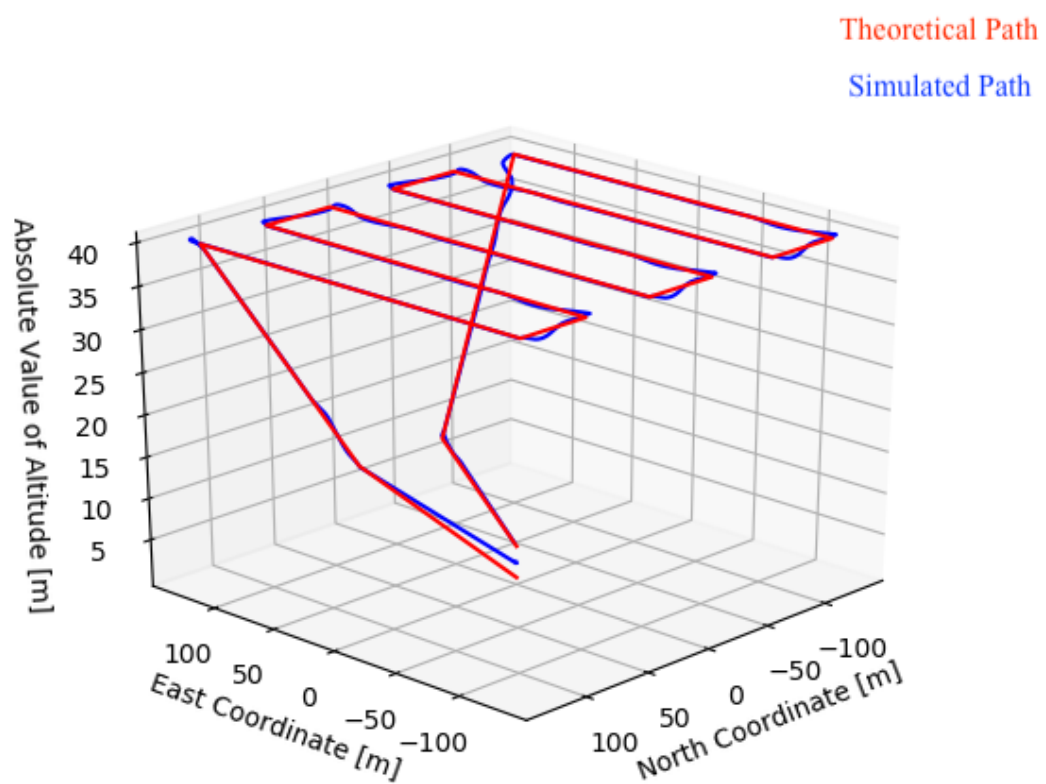


Figure 5.10 Theoretical Path - Simulated Path comparison

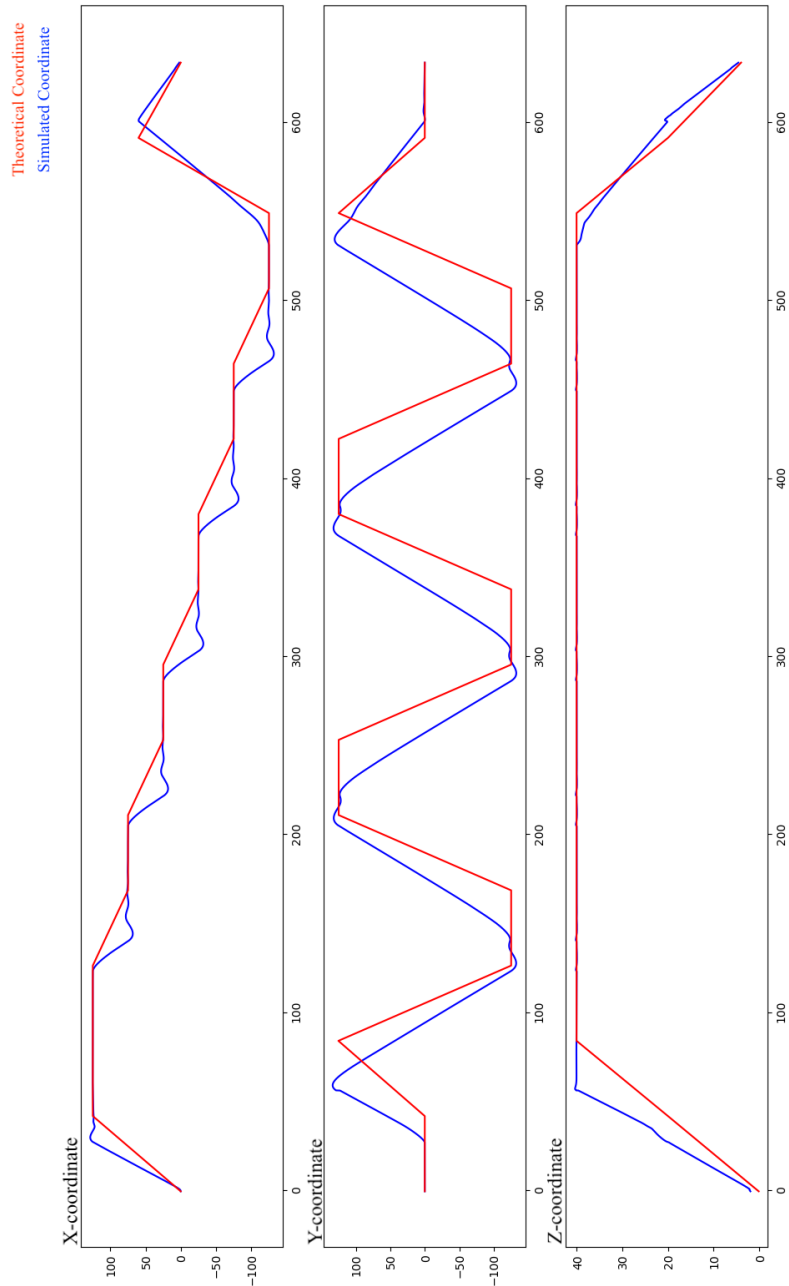


Figure 5.11 Theoretical Coordinates - Simulated Coordinates comparison

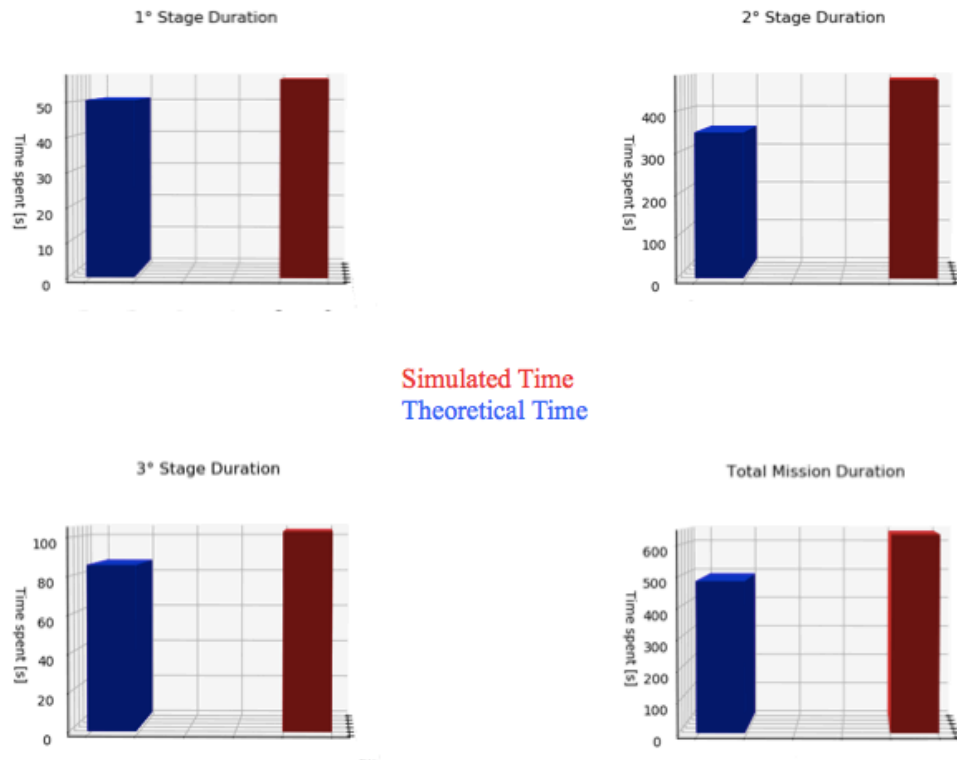


Figure 5.12 Theoretical Time - Simulated Time comparison

TIME TABLE		
	Theoretical Time [s]	Real Time [s]
1° Stage	50.59	57.16
2° Stage	349.15	485.91
3° Stage	85.94	104.83
Mission Duration	485.68	647.90

Table 5-4 Theoretical Time Values - Simulated Time Values comparison

The difference between theoretical and simulated time of the global mission is 162.22, the 33.4% more than planned duration. In sub-section 5.7 it will be given a reasonable justification for this result.

5.5. Plots and Data Second Choice Intersection Geographical FP

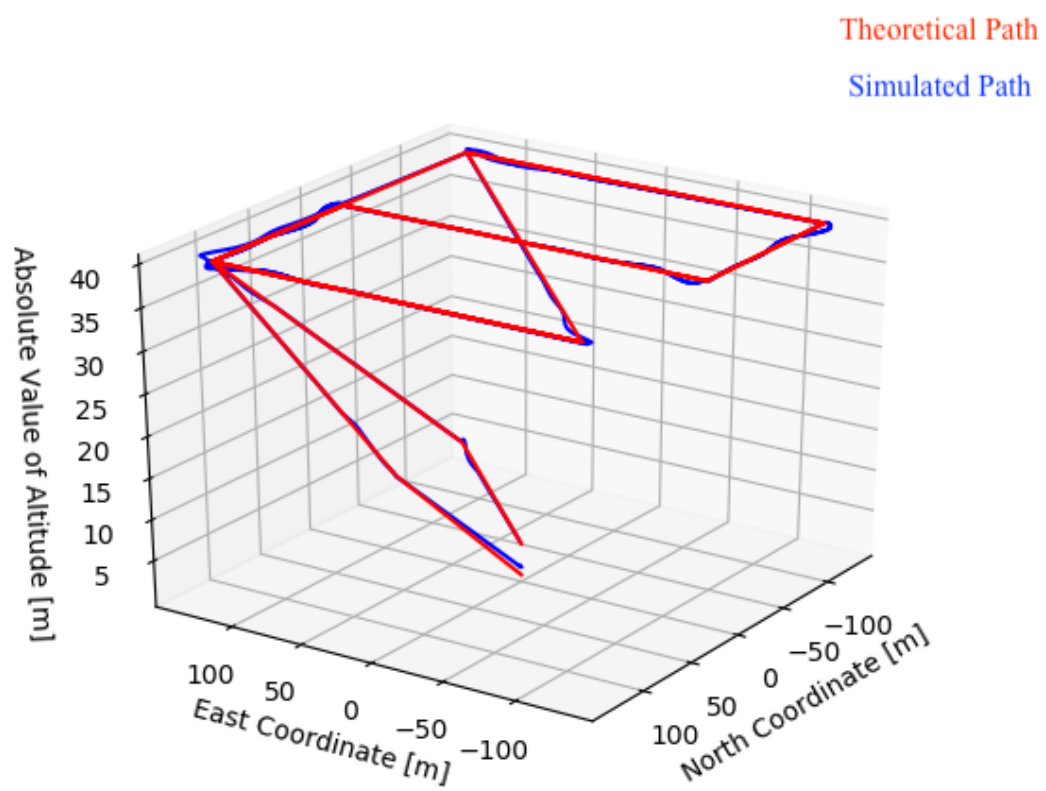


Figure 5.13 Theoretical Path - Simulated Path comparison

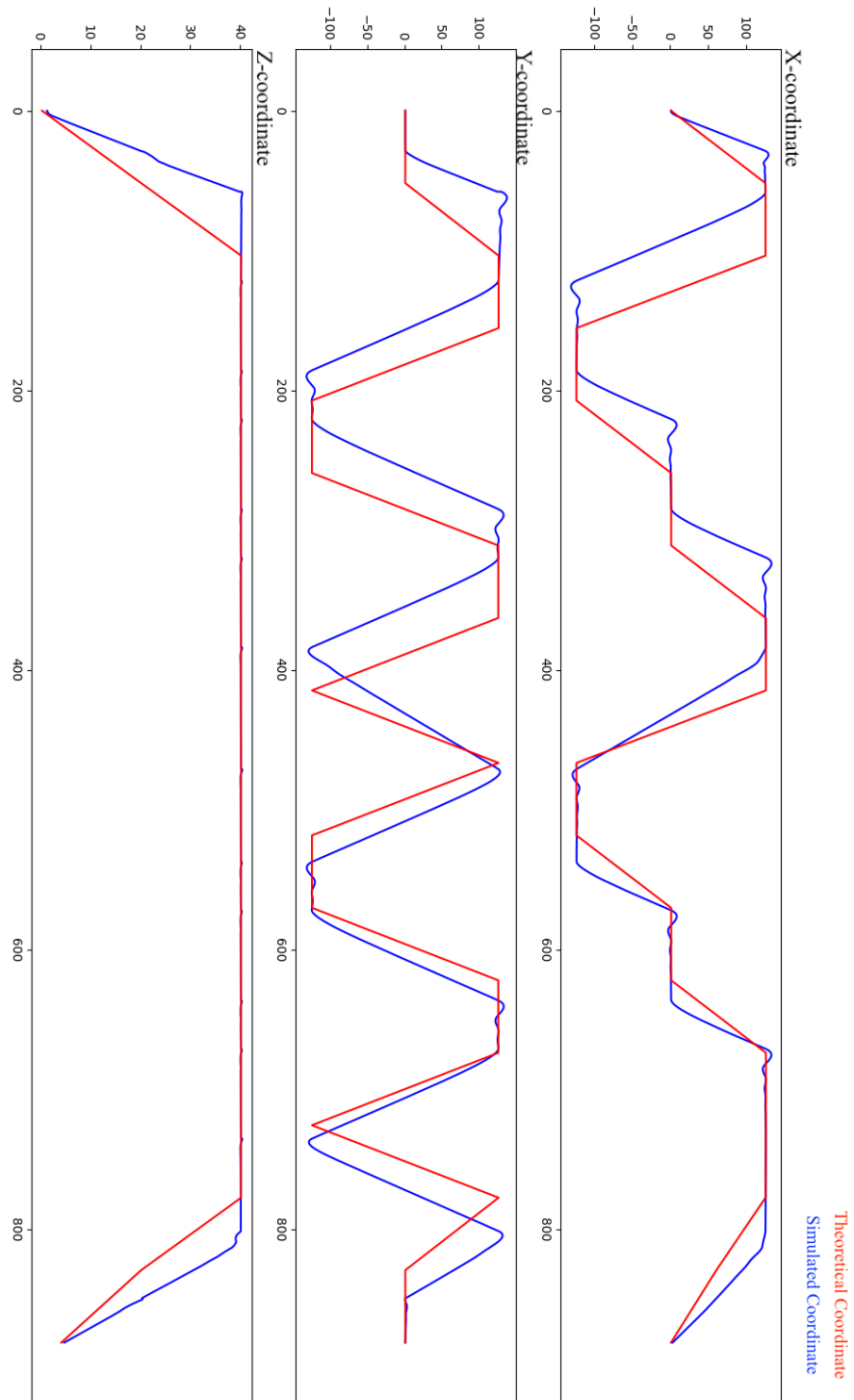


Figure 5.14 Theoretical Coordinates - Simulated Coordinates comparison

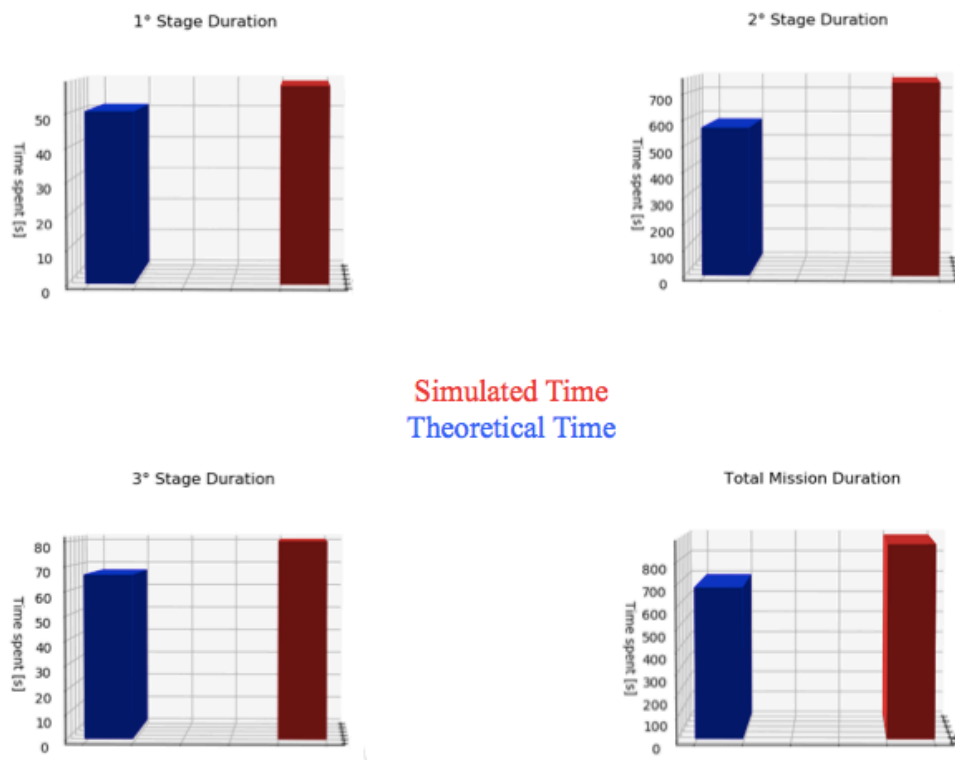


Figure 5.15 Theoretical Time - Simulated Time comparison

TIME TABLE		
	Theoretical Time [s]	Real Time [s]
1° Stage	50.59	56.44
2° Stage	570.30	742.85
3° Stage	66.77	80.16
Mission Duration	687.66	879.45

Table 5-5 Theoretical Time Values - Simulated Time Values comparison

The difference between theoretical and simulated time of the global mission is 191.79s, the 27.9% more than planned duration. In sub-section 5.7 it will be given a reasonable justification for this result.

5.6. Plots and Data Third Choice Intersection Geographical FP

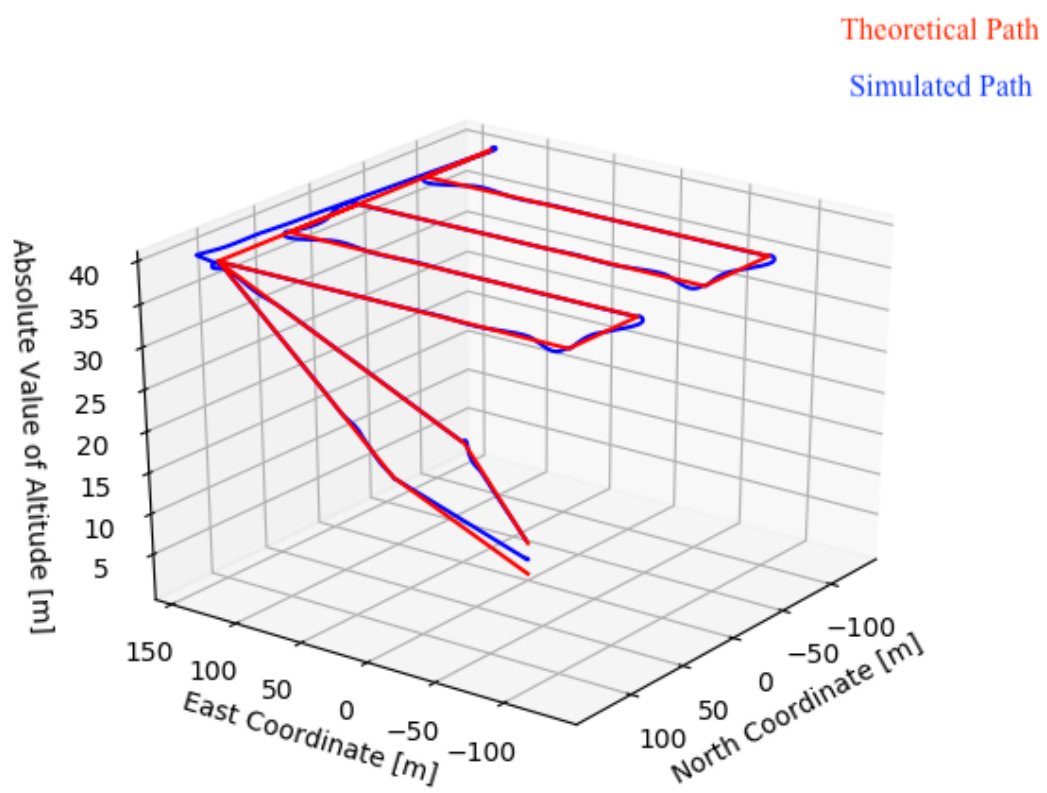


Figure 5.16 Theoretical Path - Simulated Path comparison

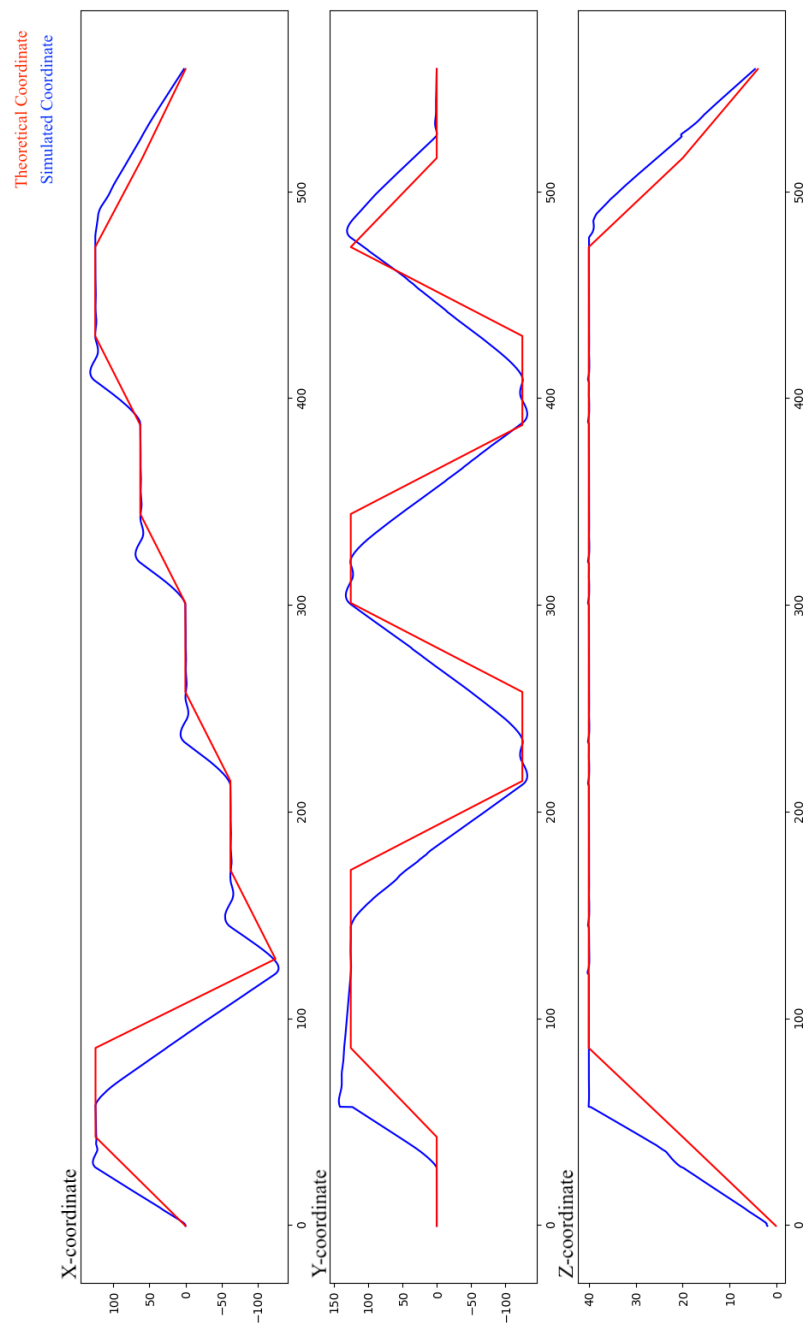


Figure 5.17 Theoretical Coordinates - Simulated Coordinates comparison

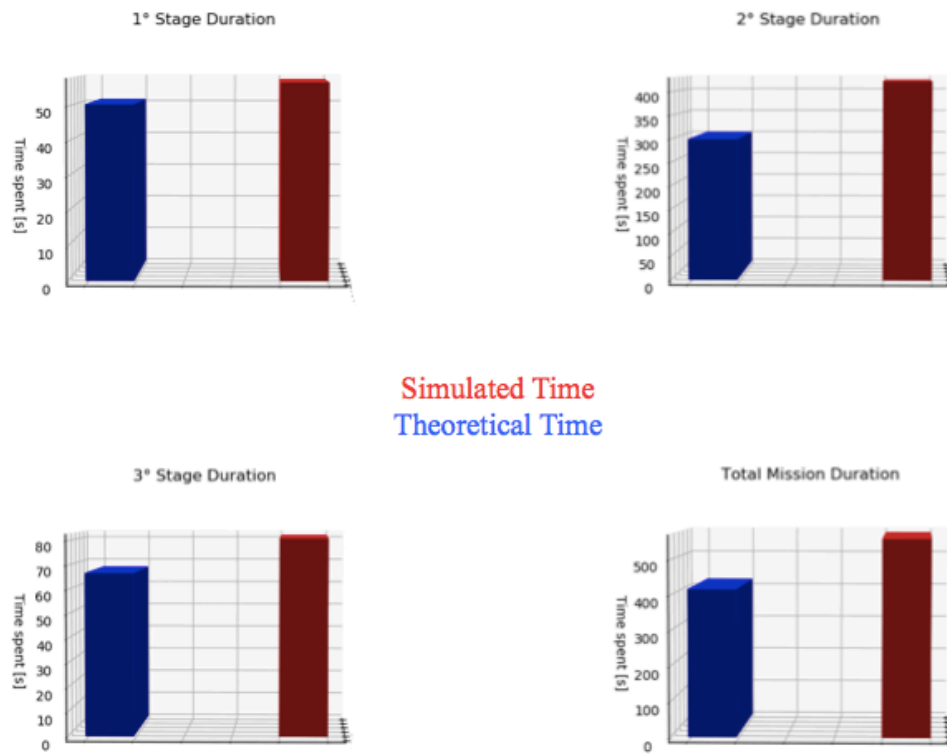


Figure 5.18 Theoretical Time - Simulated Time comparison

TIME TABLE		
	Theoretical Time [s]	Real Time [s]
1° Stage	50.59	56.54
2° Stage	299.74	404.21
3° Stage	66.76	80.15
Mission Duration	417.09	540.91

Table 5-6 Theoretical Time Values - Simulated Time Values comparison

The difference between theoretical and simulated time of the global mission is 123.82s, the 29.7% more than planned duration. In sub-section 5.7 it will be given a reasonable justification for this result.

5.7. General Comments and Results

The first plot in each section puts in evidence that *Carlo* follows more or less the theoretical path uploaded. In both first and second plots, the discrepancies (represented as oscillations in the graphs) are due to the inertia forces acting on the drone. While the drone is moving, its momentum increases. Hence, when the quadricopter reaches the desired waypoint, it has still momentum to be dissipated: this means that it will go a little bit forward before turning. Moreover, drone motion is an accelerated one, it means that the drone will accelerate and decelerate when it is close to the turning points.

To evaluate the theoretical time the uniform motion equation has been used:

$$x = x_0 + v_0 t$$

By isolating t and considering $x_0 = 0$, we obtain:

$$t = \frac{x}{v_0}$$

To evaluate the simulated time, the “time” library has been used. Before taking off, the initial time is stored with the code line (line 833) “ $t_0 = \text{time.time}()$ ”. Then, the time required to execute each stage is evaluated with the code line (line 577) inside the “stage” function “ $t_{\text{stage}} = \text{time.time}() - t_0$ ”. We can see that the maximum discrepancy between the total theoretical time and the total simulated time is quite large (33.4%). This is due to the approximation of the motion: to have a theoretical time equal to the real one we should use the same motion equations implemented in the simulator. On the contrary, since it is called “theoretical time”, it has to be evaluated with the data we have before running the simulation. The FP is all we have and inside the XML file we can only find position and velocity informations.

It has to be noticed that running the single-drone interface with the same FP, the results are every time slightly different. This is due to both simulator behaviour and machine workload. The table below presents the time values obtained for 5 different simulations of the same FP (NED FP, 1° option intersection leg).

TIME TABLE					
	1° Test	2° Test	3° Test	4° Test	5° Test
1° Stage	56.45	56.55	56.64	56.75	56.36
2° Stage	474.04	476.04	476.22	474.37	474.86
3° Stage	102.88	102.91	103.27	103.13	102.96
Mission Duration	633.37	635.50	636.13	634.25	634.18

Table 5-7 Simulated Time Values comparison of the same path

Chapter 6

CONCLUSIONS AND RECOMMENDATIONS

6.1. Preliminary Conclusions

Finally, it can be stated that the three main objectives and the extra one have been achieved. A total of six XML Flight Plans as well as the two interfaces have been implemented.

The single-drone interface allows *Carlo* to execute its FP containing four different kind of legs; moreover, six significant graphs can be plotted at the end of each simulation to analyse drone behaviour.

The multiple-drones interface allows *Paolo* and *Francesca* to simultaneously complete their respective FP. A video of the simulation can be found on Youtube via this link: <https://youtu.be/ml-OH3kf6lo>

All FPs and both Interfaces could be found in GitHub website via this link: <https://github.com/Francesco-Rose/Python-Interfaces-for-AirSim>

6.2. Area for further studies

Neighborhood-Surveillance-Mission is aimed to monitor the chosen AirSim Environment. In the real world, the organization of a mission like this points out several issues that have to be deeply analyzed.

First of all, we have to think to the respect for privacy. It is clear that people do not want a drone equipped with a camera flying out of their houses. It should be created a set of rules that do not allow drones to take photos or videos inside houses; the drone should be able to only monitor the external zone.

Secondly, aerial traffic management should be taken into account. In densely populated areas, a collision between two or more drones could have disastrous effects. Moreover, we have also to consider possible collisions with buildings and other obstacles as well as the running out of batteries. Emergency and Contingency Flight Plans have to be implemented to allow drones safe flight.

Another important aspect that has to be pondered is the need for “good” drones. The more this technology improves, the more evil-minded people take advantages of drones for bad purposes. Hence, the presence of police-drones is becoming more and more important to ensure population safety from any kind of danger.

Finally, drones operating in a real environment should be able to adapt themselves to any kind of change of the physical parameters (air pressure, air speed, temperature, air flow, etc). Moreover, wheater conditions must not interfere with both drone performances and the development of the mission.

6.3. Recommendations

This section wants to point out all the “rules and limitations” to use both dynamic interfaces. Some of the limitations refer to the legs functions and to the legs implementation in the XML file; rest of limitations refer to other part of the code.

6.3.1. Recommendations for both interfaces

The following list contains the recommendations that have to be followed to correctly use both interfaces.

1. Along the handwriting of the XML files, it should not be left any empty space at the beginning and at the end of the text of both “<Nextlist>” (intersection leg) and “<body>” (iterative leg). White spaces are only allowed to separate the name of the points inside these lists.
2. In the XML file, for a *scan leg*, the points tags that delimit the scan area have to be named as “<point1>”, “<point2>”, “<point3>”, “<point4>” and not as “<point>” (RAISE+ build the *scan leg* with <point>). Moreover, the attribute’s name and the attribute’s value that identify the leg type have to be written in the form “xsi_type = fp_TFLeg” and not as “xsi:type = fp:TFLeg”. This is due to the presence of colons that is troublesome to Python while searching for the leg’s type in the parsed XML.
3. The organization of the scan path has to be carefully implemented. The scan area has to be a square centered in the reference-frame-system origin or a square corresponding to one of the four quadrants formed by the Cartesian plane; its sides have to be parallel to the NED axis. Moreover, the starting point coordinates of the scan have to be written inside the “<point1>” element. It has also to be noticed that the short section of the scan will follow the x-axis.
4. The dynamic interface as well as all its functions is independent from the number of stages, number of legs, waypoints selected, etc. Take off and landing are automatically executed and the landing point will be the same of the take off one. To implement the FP, it has to be remembered that we can use only the four legs described in section 2.4.2.

6.3.2. Recommendations for Single-Drone interface

To perfectly execute landing, the last point of the FP has to be above the take off point. This is due to the utilization of the “moveByVelocityZAsync” function that allows a vertical movement, increasing or decreasing the altitude.

Figure 6.1 shows the AirSim’s file “settings.json” to be used for the single-drone interface.

```
{
  "SeeDocsAt": "https://github.com/Microsoft/AirSim/blob/master/docs/settings.md",
  "SettingsVersion": 1.2,
  "SimMode": "Multirotor"
}
```

Figure 6.1 Setting.json for Single-Drone Interface

6.3.3. Recommendations for Multiple-Drones interface

The following list contains the recommendations that have to be followed to correctly use the multiple-drones interface.

1. The FPs designer has to carefully select the waypoints covered by the drones to avoid collisions between drones or with obstacles.
2. Figure 6.2 shows the AirSim's file "settings.json" to be used for the multiple-drones interface. The starting point of each drone can be manually set inserting the desired NED coordinates.

```
{
  "SeeDocsAt": "https://github.com/Microsoft/AirSim/blob/master/docs/settings.md",
  "SettingsVersion": 1.2,
  "SimMode": "Multirotor",

  "Vehicles": {
    "Drone1": {
      "VehicleType": "SimpleFlight",
      "X": 0, "Y": 0, "Z": -2,
      "Yaw": -180
    },
    "Drone2": {
      "VehicleType": "SimpleFlight",
      "X": 2, "Y": 0, "Z": -2
    }
  }
}
```

Figure 6.2 Setting.json for Multiple-Drones Interface

3. The multiple-drones interface does not allow us to put an Iterative Leg or an Intersection Leg in the path to be iterated for an iterative leg; the multiple-drones interface does not allow us to put an Iterative Leg or an Intersection Leg as possible choice to be selected for an intersection leg.
4. It has to be implemented at least one FP for each drone that is going to be used.

Chapter 7

BIBLIOGRAPHY

The bibliography has been built with “Mendeley Desktop” Software.

- [1] 19.7. xml.etree.ElementTree — The ElementTree XML API — Python 2.7.16 documentation. (n.d.). Retrieved October 2, 2019, from <https://docs.python.org/2/library/xml.etree.elementtree.html>
- [2] 20.5. xml.etree.ElementTree — The ElementTree XML API — Python 3.5.7 documentation. (n.d.). Retrieved October 2, 2019, from <https://docs.python.org/3.5/library/xml.etree.elementtree.html#xml.etree.ElementTree.Element>
- [3] A Brief History of Markup & XML. (n.d.). Retrieved October 2, 2019, from <https://codepunk.io/a-brief-history-of-markup-and-xml/>
- [4] AirSim APIs — airsim documentation. (n.d.). Retrieved October 2, 2019, from <https://airsim-fork.readthedocs.io/en/docs/apis.html>
- [5] Allen, J. D., & Unicode Consortium. (2007). *The Unicode standard 5.0*. Addison-Wesley. Retrieved from <https://www.w3.org/TR/xml/>
- [6] An easy introduction to 3D plotting with Matplotlib. (n.d.). Retrieved October 5, 2019, from <https://towardsdatascience.com/an-easy-introduction-to-3d-plotting-with-matplotlib-801561999725>
- [7] An Intro to Threading in Python – Real Python. (n.d.). Retrieved November 22, 2019, from <https://realpython.com/intro-to-python-threading/>
- [8] Beri, M. (2018). *Python 3*. (F. Brivio, Ed.). Monza (MI): Apogeo.
- [9] Come lavorare con i Documenti di Excel usando Python. (n.d.). Retrieved October 8, 2019, from <https://code.tutsplus.com/it/tutorials/how-to-work-with-excel-documents-using-python--cms-25698>
- [10] Illera, T. M. (2014). Flight Plan Documentation, (March), 0–120.
- [11] Image APIs - AirSim. (n.d.). Retrieved October 2, 2019, from https://microsoft.github.io/AirSim/docs/image_apis/
- [12] Processing XML in Python with ElementTree - Eli Bendersky's website. (n.d.). Retrieved October 2, 2019, from <https://eli.thegreenplace.net/2012/03/15/processing-xml-in-python-with-elementtree/>
- [13] Python – Read XML file (DOM Example) – Mkyong.com. (n.d.). Retrieved October 2, 2019, from <https://www.mkyong.com/python/python-read-xml-file-dom-example/>
- [14] Python time clock() Method - Tutorialspoint. (n.d.). Retrieved October 5, 2019, from https://www.tutorialspoint.com/python/time_clock.htm
- [15] Python XML with ElementTree (article) - DataCamp. (n.d.). Retrieved October 2, 2019, from <https://www.datacamp.com/community/tutorials/python-xml-elementtree>
- [16] Tutorial 1: Create a simple XLSX file — XlsxWriter Documentation. (n.d.). Retrieved October 8, 2019, from <https://xlsxwriter.readthedocs.io/tutorial01.html>
- [17] Unreal Engine. (n.d.). Retrieved from https://en.wikipedia.org/wiki/Unreal_Engine
- [18] Welcome to AirSim — airsim documentation. (n.d.). Retrieved October 2, 2019, from <https://airsim-fork.readthedocs.io/en/docs/#>
- [19] What is Unreal Engine 4. (n.d.). Retrieved October 2, 2019, from

- <https://www.unrealengine.com/en-US/what-is-unreal-engine-4>
- [20] World Wide Web Consortium. (n.d.). Retrieved from https://en.wikipedia.org/wiki/World_Wide_Web_Consortium
- [21] XML. (n.d.). Retrieved from <https://en.wikipedia.org/wiki/XML>
- [22] XML Syntax. (n.d.). Retrieved October 2, 2019, from https://www.w3schools.com/xml/xml_syntax.asp
- [23] XPath Support in ElementTree. (n.d.). Retrieved October 2, 2019, from <http://effbot.org/zone/element-xpath.htm>

ANNEX

All FPs and both Interfaces could be found in GitHub website via this link:

<https://github.com/Francesco-Rose/Python-Interfaces-for-AirSim>

- **SINGLE DRONE'S FLIGHT PLAN (NED Coordinates).**

```
<FlightPlan>
  <Locale>
    <speedUnits>ms</speedUnits>
    <altitudeUnits>m</altitudeUnits>
    <distanceUnits>m</distanceUnits>
    <decimalSeparator>.</decimalSeparator>
    <groupSeparator/>
  </Locale>

  <MainFP>
    <name>NEIGHBORHOOD FLIGHT PLAN.</name>
    <description>Neighborhood Surveillance Mission.</description>

    <stages>
      <stage id="First Part">
        <leg id="zero-point" xsi_type="fp_TFLeg">
          <dest>
            <name>N0</name>
            <north_coordinate>125</north_coordinate>
            <east_coordinate>0</east_coordinate>
            <altitude>-20</altitude>
            <speed>5</speed>
            <next>first-point</next>
          </dest>
        </leg>
        <leg id="first-point" xsi_type="fp_TFLeg">
          <dest>
            <name>N1</name>
            <north_coordinate>125</north_coordinate>
            <east_coordinate>125</east_coordinate>
            <altitude>-40</altitude>
            <speed>5</speed>
            <next>second-point</next>
          </dest>
        </leg>
      </stage>
    </stages>
  </MainFP>
</FlightPlan>
```

```

<stage id="Intersection Leg Part">
  <leg id="second-point" xsi_type="fp_IntersectionLeg">
    <nextList>third-point-a third-point-b third-point-c</nextList>
  </leg>
  <leg id="third-point-a" xsi_type="fp_Scan">
    <dest>
      <coordinates>0 0</coordinates>
    </dest>
    <trackseparation>50</trackseparation>
    <area>
      <point1>125 -125</point1>
      <point2>-125 -125</point2>
      <point3>-125 125</point3>
      <point4>125 125</point4>
    </area>
    <speed>5</speed>
    <altitude>-40</altitude>
  </leg>
  <leg id="third-point-b" xsi_type="fp_IterativeLeg">
    <next>fourth-point</next>
    <body>third-point-b-one third-point-b-two</body>
    <upperBound>2</upperBound>
    <first>third-point-b-one</first>
    <last>third-point-b-two</last>
  </leg>
  <leg id="third-point-b-one" xsi_type="fp_TFLeg">
    <dest>
      <name>N3B1</name>
      <north_coordinate>-125</north_coordinate>
      <east_coordinate>125</east_coordinate>
      <altitude>-40</altitude>
      <speed>5</speed>
      <next>Home1</next>
    </dest>
  </leg>

```

```

<leg id="third-point-b-two" xsi_type="fp_Scan">
  <dest>
    <coordinates>0 0</coordinates>
  </dest>
  <trackseparation>125</trackseparation>
  <area>
    <point1>-125 -125</point1>
    <point2>-125 -125</point2>
    <point3>125 -125</point3>
    <point4>125 125</point4>
  </area>
  <speed>5</speed>
  <altitude>-40</altitude>
</leg>
<leg id="fourth-point" xsi_type="fp_TFLeg">
  <dest>
    <name>N4</name>
    <north_coordinate>125</north_coordinate>
    <east_coordinate>125</east_coordinate>
    <altitude>-40</altitude>
    <speed>5</speed>
    <next>Home1</next>
  </dest>
</leg>
<leg id="third-point-c" xsi_type="fp_Scan">
  <dest>
    <coordinates>0 0</coordinates>
  </dest>
  <trackseparation>62.5</trackseparation>
  <area>
    <point1>-125 125</point1>
    <point2>-125 -125</point2>
    <point3>125 -125</point3>
    <point4>125 125</point4>
  </area>
  <speed>5</speed>
  <altitude>-40</altitude>
</leg>
<initialLegs> second-point </initialLegs>
<finalLegs> third-point-a third-point-b third-point-c </finalLegs>
</stage>

```



```
<stage id="Go Home Part">
  <leg id="Home1" xsi_type="fp_TFLeg">
    <dest>
      <name>Home1</name>
      <north_coordinate>60</north_coordinate>
      <east_coordinate>0</east_coordinate>
      <altitude>-20</altitude>
      <speed>4</speed>
      <next>Home2</next>
    </dest>
  </leg>
  <leg id="Home2" xsi_type="fp_TFLeg">
    <dest>
      <name>Home2</name>
      <north_coordinate>0</north_coordinate>
      <east_coordinate>0</east_coordinate>
      <altitude>-4</altitude>
      <speed>2</speed>
    </dest>
  </leg>
</stage>

</stages>

</MainFP>

</FlightPlan>
```

- **SINGLE DRONE'S FLIGHT PLAN (Geographical Coordinates)**

```
<FlightPlan>
  <Locale>
    <speedUnits>ms</speedUnits>
    <altitudeUnits>m</altitudeUnits>
    <trackseparationUnits>m</trackseparationUnits>
    <distanceUnits>°</distanceUnits>
    <decimalSeparator>.</decimalSeparator>
    <groupSeparator/>
  </Locale>

  <MainFP>
    <name>NEIGHBORHOOD FLIGHT PLAN.</name>
    <description>Neighborhood Surveillance Mission.</description>

    <stages>
      <stage id="First Part">
        <leg id="zero-point" xsi_type="fp_TFLeg">
          <dest>
            <name>N0</name>
            <latitude>47.64260464285712</latitude>
            <longitude>-122.140365</longitude>
            <altitude>143.199297198</altitude>
            <speed>5</speed>
            <next>first-point</next>
          </dest>
        </leg>
        <leg id="first-point" xsi_type="fp_TFLeg">
          <dest>
            <name>N1</name>
            <latitude>47.64260464285712</latitude>
            <longitude>-122.1386985308925</longitude>
            <altitude>163.199297198</altitude>
            <speed>5</speed>
            <next>second-point</next>
          </dest>
        </leg>
      </stage>
    </stages>
  </MainFP>
</FlightPlan>
```

```

<stage id="Intersection Leg Part">
  <leg id="second-point" xsi_type="fp_IntersectionLeg">
    <nextList>third-point-a third-point-b third-point-c</nextList>
  </leg>
  <leg id="third-point-a" xsi_type="fp_Scan">
    <dest>
      <coordinates>0 0</coordinates>
    </dest>
    <trackseparation>50</trackseparation>
    <area>
      <point1>47.64260464285712 -122.14203146910751</point1>
      <point2>47.640360097142874 -122.14203146910751</point2>
      <point3>47.640360097142874 -122.1386985308925</point3>
      <point4>47.64260464285712 -122.1386985308925</point4>
    </area>
    <speed>5</speed>
    <altitude>163.199297198</altitude>
  </leg>
  <leg id="third-point-b" xsi_type="fp_IterativeLeg">
    <next>fourth-point</next>
    <body>third-point-b-one third-point-b-two</body>
    <upperBound>2</upperBound>
    <first>third-point-b-one</first>
    <last>third-point-b-two</last>
  </leg>
  <leg id="third-point-b-one" xsi_type="fp_TFLeg">
    <dest>
      <name>N3B1</name>
      <latitude>47.640360097142874</latitude>
      <longitude>-122.1386985308925</longitude>
      <altitude>163.199297198</altitude>
      <speed>5</speed>
      <next>Home1</next>
    </dest>
  </leg>

```

```

<leg id="third-point-b-two" xsi_type="fp_Scan">
  <dest>
    <coordinates>0 0</coordinates>
  </dest>
  <trackseparation>125</trackseparation>
  <area>
    <point1>47.640360097142874 -122.14203146910751</point1>
    <point2>47.640360097142874 -122.1386985308925</point2>
    <point3>47.64260464285712 -122.14203146910751</point3>
    <point4>47.64260464285712 -122.1386985308925</point4>
  </area>
  <speed>5</speed>
  <altitude>163.199297198</altitude>
</leg>
<leg id="fourth-point" xsi_type="fp_TFLeg">
  <dest>
    <name>N4</name>
    <latitude>47.64260464285712</latitude>
    <longitude>-122.1386985308925</longitude>
    <altitude>163.199297198</altitude>
    <speed>5</speed>
    <next>Home1</next>
  </dest>
</leg>
<leg id="third-point-c" xsi_type="fp_Scan">
  <dest>
    <coordinates>0 0</coordinates>
  </dest>
  <trackseparation>62.5</trackseparation>
  <area>
    <point1>47.640360097142874 -122.1386985308925</point1>
    <point2>47.640360097142874 -122.14203146910751</point2>
    <point3>47.64260464285712 -122.14203146910751</point3>
    <point4>47.64260464285712 -122.1386985308925</point4>
  </area>
  <speed>5</speed>
  <altitude>163.199297198</altitude>
</leg>
<initialLegs> second-point </initialLegs>
<finalLegs> third-point-a third-point-b third-point-c </finalLegs>
</stage>

```

```
<stage id="Go Home Part">
  <leg id="Home1" xsi_type="fp_TFLeg">
    <dest>
      <name>Home1</name>
      <latitude>47.642021060971416</latitude>
      <longitude>-122.140365</longitude>
      <altitude>143.199297198</altitude>
      <speed>4</speed>
      <next>Home2</next>
    </dest>
  </leg>
  <leg id="Home2" xsi_type="fp_TFLeg">
    <dest>
      <name>Home2</name>
      <latitude>47.64148237</latitude>
      <longitude>-122.140365</longitude>
      <altitude>127.199297198</altitude>
      <speed>2</speed>
    </dest>
  </leg>
</stage>

</stages>

</MainFP>

</FlightPlan>
```

- SINGLE DRONE'S PYTHON CODE

```
1  # Author of the Code: Francesco Rose
2  # Master Thesis Project, Universitat Politecnica de la Catalunya
3  # Dynamic Interface AirSim/XML Fight Plan
4
5  import airsims
6
7  import pprint
8  import time
9  from math import *
10
11 import xlswriter
12
13 import xml.etree.ElementTree as ET
14
15 from mpl_toolkits import mplot3d
16 import numpy as np
17 import matplotlib.pyplot as plt
18 import openpyxl
19
20 from geographiclib import geodesic
21
22
23 # ----- Creation of the Functions -----
24
25 # Get_Data_to_Plot_drone's_track Function
26
27 def get_data(state):
28     global column
29     column += 1
30     n_c = state.kinematics_estimated.position.x_val
31     e_c = state.kinematics_estimated.position.y_val
32     h = state.kinematics_estimated.position.z_val
33     worksheet.write(0, column, n_c)
34     worksheet.write(1, column, e_c)
35     worksheet.write(2, column, h)
36
37
```

```

38 # Check_Position Function
39
40 def check_position(list1):
41     state = client.getMultirotorState()
42     p_attuale = [state.kinematics_estimated.position.x_val, state.kinematics_estimated.position.y_val,
43                 state.kinematics_estimated.position.z_val]
44     distance = sqrt((list1[0] - p_attuale[0])**2 + (list1[1] - p_attuale[1])**2 + (list1[2] - p_attuale[2])**2)
45     time_required = (distance / list1[3])
46     Time_For_Each_Leg.append(time_required)
47     if LOG == 'ON':
48         a = True
49         while a:
50             if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < p_attuale[1] < (list1[1] + 2) and \
51                 (list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
52                 a = False
53             else:
54                 get_data(state)
55                 time.sleep(0.1)
56                 state = client.getMultirotorState()
57                 p_attuale = [state.kinematics_estimated.position.x_val, state.kinematics_estimated.position.y_val,
58                             state.kinematics_estimated.position.z_val]
59         else:
60             time.sleep(time_required*0.9)
61             a = True
62             while a:
63                 if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < p_attuale[1] < (list1[1] + 2) and \
64                     (list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
65                     a = False
66                 else:
67                     time.sleep(0.1)
68                     state = client.getMultirotorState()
69                     p_attuale = [state.kinematics_estimated.position.x_val, state.kinematics_estimated.position.y_val,
70                                 state.kinematics_estimated.position.z_val]
71
72 # To_Fix_Leg Function
73
74 def tf_leg(n_coord, e_coord, alt, speed):
75     return client.moveToPositionAsync(n_coord, e_coord, alt, speed)
76
77
78

```

```

79 # Scan_Leg Function
80
81 def scan_leg(ts, f_p_n, f_p_e, v, h, i):
82     client.moveToPositionAsync(f_p_n, f_p_e, h, v)
83     list = [f_p_n, f_p_e, h, v]
84     North_Coordinates2.append(list[0])
85     East_Coordinates2.append(list[1])
86     Altitudes2.append(list[2])
87     Speeds.append(list[3])
88     check_position(list)
89     if (f_p_e < 0 < f_p_n) or (f_p_n > 0 and f_p_e == 0):
90         if i % 2 == 0:
91             p = []
92             s = []
93             t = []
94             q = []
95             for num in range(1, i, 2):
96                 sott1 = num * ts
97                 a = f_p_n - sott1
98                 p = (a, f_p_e, h, v)
99                 p.append(p)
100                 s = (a, f_p_e + 11, h, v)
101                 s.append(s)
102             for mol in range(2, i+1, 2):
103                 sott2 = mol * ts
104                 b = f_p_n - sott2
105                 T = (b, f_p_e + 11, h, v)
106                 t.append(T)
107                 Q = (b, f_p_e, h, v)
108                 q.append(Q)
109             Total = []
110             for n in range(len(q)):
111                 total = [p[n], s[n], t[n], q[n]]
112                 Total.append(total)
113             Totale = []
114             for n in range(len(Total)):
115                 for a in range(4):
116                     totale = Total[n][a]
117                     Totale.append(totale)
118             print(Totale)
119             for n in range(len(Totale)):
120                 client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3])
121                 list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
122                 North_Coordinates2.append(list1[0])

```



```

123         East_Coordinates2.append(list1[1])
124         Altitudes2.append(list1[2])
125         Speeds.append(list1[3])
126         check_position(list1)
127     else:
128         p = []
129         s = []
130         t = []
131         q = []
132         for num in range(1, i+1, 2):
133             sott1 = num * ts
134             a = f_p_n - sott1
135             P = (a, f_p_e, h, v)
136             p.append(P)
137             S = (a, f_p_e + ll, h, v)
138             s.append(S)
139         for mol in range(2, i+2, 2):
140             sott2 = mol * ts
141             b = f_p_n - sott2
142             T = (b, f_p_e + ll, h, v)
143             t.append(T)
144             Q = (b, f_p_e, h, v)
145             q.append(Q)
146         Total = []
147         for n in range(len(s)):
148             total = [p[n], s[n], t[n], q[n]]
149             Total.append(total)
150         Totale = []
151         for n in range(len(Total)):
152             for a in range(4):
153                 totale = Total[n][a]
154                 Totale.append(totale)
155         print(Totale)
156         limit = len(Totale)-2
157         for n in range(limit):
158             client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3])
159             list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
160             North_Coordinates2.append(list1[0])
161             East_Coordinates2.append(list1[1])
162             Altitudes2.append(list1[2])
163             Speeds.append(list1[3])
164             check_position(list1)
165     elif (f_p_n > 0 and f_p_e > 0) or (f_p_n == 0 and f_p_e > 0):
166         if i % 2 == 0:

```

```

167     p = []
168     s = []
169     t = []
170     q = []
171     for num in range(1, i, 2):
172         sott1 = num * ts
173         a = f_p_n - sott1
174         P = (a, f_p_e, h, v)
175         p.append(P)
176         S = (a, f_p_e - ll, h, v)
177         s.append(S)
178     for mol in range(2, i+1, 2):
179         sott2 = mol * ts
180         b = f_p_n - sott2
181         T = (b, f_p_e - ll, h, v)
182         t.append(T)
183         Q = (b, f_p_e, h, v)
184         q.append(Q)
185     Total = []
186     for n in range(len(q)):
187         total = [p[n], s[n], t[n], q[n]]
188         Total.append(total)
189     Totale = []
190     for n in range(len(Total)):
191         for a in range(4):
192             totale = Total[n][a]
193             Totale.append(totale)
194     print(Totale)
195     for n in range(len(Totale)):
196         client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3])
197         list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
198         North_Coordinates2.append(list1[0])
199         East_Coordinates2.append(list1[1])
200         Altitudes2.append(list1[2])
201         Speeds.append(list1[3])
202         check_position(list1)
203     else:
204         p = []
205         s = []
206         t = []
207         q = []
208         for num in range(1, i+1, 2):
209             sott1 = num * ts
210             a = f_p_n - sott1

```

```

211         p = (a, f_p_e, h, v)
212         p.append(P)
213         s = (a, f_p_e - ll, h, v)
214         s.append(S)
215         for mol in range(2, i+2, 2):
216             sott2 = mol * ts
217             b = f_p_n - sott2
218             T = (b, f_p_e - ll, h, v)
219             t.append(T)
220             Q = (b, f_p_e, h, v)
221             q.append(Q)
222         Total = []
223         for n in range(len(s)):
224             total = [p[n], s[n], t[n], q[n]]
225             Total.append(total)
226         Totale = []
227         for n in range(len(Total)):
228             for a in range(4):
229                 totale = Total[n][a]
230                 Totale.append(totale)
231         print(Totale)
232         limit = len(Totale) - 2
233         for n in range(limit):
234             client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3])
235             list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
236             North_Coordinates2.append(list1[0])
237             East_Coordinates2.append(list1[1])
238             Altitudes2.append(list1[2])
239             Speeds.append(list1[3])
240             check_position(list1)
241     elif (f_p_n < 0 < f_p_e) or (f_p_n < 0 and f_p_e == 0):
242         if i % 2 == 0:
243             p = []
244             s = []
245             t = []
246             q = []
247             for num in range(1, i, 2):
248                 sott1 = num * ts
249                 a = f_p_n + sott1
250                 P = (a, f_p_e, h, v)
251                 p.append(P)
252                 S = (a, f_p_e - ll, h, v)
253                 s.append(S)
254             for mol in range(2, i+1, 2):

```

```

255         sott2 = mol * ts
256         b = f_p_n + sott2
257         T = (b, f_p_e - ll, h, v)
258         t.append(T)
259         Q = (b, f_p_e, h, v)
260         q.append(Q)
261     Total = []
262     for n in range(len(q)):
263         total = [p[n], s[n], t[n], q[n]]
264         Total.append(total)
265     Totale = []
266     for n in range(len(Total)):
267         for a in range(4):
268             totale = Total[n][a]
269             Totale.append(totale)
270     print(Totale)
271     for n in range(len(Totale)):
272         client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3])
273         list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
274         North_Coordinates2.append(list1[0])
275         East_Coordinates2.append(list1[1])
276         Altitudes2.append(list1[2])
277         Speeds.append(list1[3])
278         check_position(list1)
279     else:
280         p = []
281         s = []
282         t = []
283         q = []
284         for num in range(1, i+1, 2):
285             sott1 = num * ts
286             a = f_p_n + sott1
287             P = (a, f_p_e, h, v)
288             p.append(P)
289             S = (a, f_p_e - ll, h, v)
290             s.append(S)
291         for mol in range(2, i+2, 2):
292             sott2 = mol * ts
293             b = f_p_n + sott2
294             T = (b, f_p_e - ll, h, v)
295             t.append(T)
296             Q = (b, f_p_e, h, v)
297             q.append(Q)
298         Total = []

```

```

299         for n in range(len(s)):
300             total = [p[n], s[n], t[n], q[n]]
301             Total.append(total)
302         Totale = []
303         for n in range(len(Total)):
304             for a in range(4):
305                 totale = Total[n][a]
306                 Totale.append(totale)
307         limit = len(Totale) - 2
308         for n in range(limit):
309             client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3])
310             list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
311             North_Coordinates2.append(list1[0])
312             East_Coordinates2.append(list1[1])
313             Altitudes2.append(list1[2])
314             Speeds.append(list1[3])
315             check_position(list1)
316     elif (f_p_n < 0 and f_p_e < 0) or (f_p_n == 0 and f_p_e < 0):
317         if i % 2 == 0:
318             p = []
319             s = []
320             t = []
321             q = []
322             for num in range(1, i, 2):
323                 sott1 = num * ts
324                 a = f_p_n + sott1
325                 P = (a, f_p_e, h, v)
326                 p.append(P)
327                 S = (a, f_p_e + ll, h, v)
328                 s.append(S)
329             for mol in range(2, i+1, 2):
330                 sott2 = mol * ts
331                 b = f_p_n + sott2
332                 T = (b, f_p_e + ll, h, v)
333                 t.append(T)
334                 Q = (b, f_p_e, h, v)
335                 q.append(Q)
336             Total = []
337             for n in range(len(q)):
338                 total = [p[n], s[n], t[n], q[n]]
339                 Total.append(total)
340             Totale = []
341             for n in range(len(Total)):
342                 for a in range(4):

```

```

343         totale = Total[n][a]
344         Totale.append(totale)
345     print(Totale)
346     for n in range(len(Totale)):
347         client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3])
348         list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
349         North_Coordinates2.append(list1[0])
350         East_Coordinates2.append(list1[1])
351         Altitudes2.append(list1[2])
352         Speeds.append(list1[3])
353         check_position(list1)
354     else:
355         p = []
356         s = []
357         t = []
358         q = []
359         for num in range(1, i+1, 2):
360             sott1 = num * ts
361             a = f_p_n + sott1
362             P = (a, f_p_e, h, v)
363             p.append(P)
364             S = (a, f_p_e + ll, h, v)
365             s.append(S)
366         for mol in range(2, i+2, 2):
367             sott2 = mol * ts
368             b = f_p_n + sott2
369             T = (b, f_p_e + ll, h, v)
370             t.append(T)
371             Q = (b, f_p_e, h, v)
372             q.append(Q)
373         Total = []
374         for n in range(len(s)):
375             total = [p[n], s[n], t[n], q[n]]
376             Total.append(total)
377         Totale = []
378         for n in range(len(Total)):
379             for a in range(4):
380                 totale = Total[n][a]
381                 Totale.append(totale)
382     print(Totale)
383     limit = len(Totale) - 2
384     for n in range(limit):
385         client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3])
386         list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]

```

```

387         North_Coordinates2.append(list1[0])
388         East_Coordinates2.append(list1[1])
389         Altitudes2.append(list1[2])
390         Speeds.append(list1[3])
391         check_position(list1)
392
393
394     # Iterative_Leg Function
395
396     def iterative_leg(name_point, num_of_iter, name_next_point):
397         l = waypoints_data[name_point]["len_iter_path"]
398         iter_path = []
399         for n in range(l):
400             iter_path.append(waypoints_data[name_point]["name_point_iter_path" + str(n)])
401         rip = 0
402         while rip < num_of_iter:
403             for n in range(len(iter_path)):
404                 name_point = iter_path[n]
405                 if waypoints_data[name_point]["leg_type"] == "fp_TFLeg":
406                     n_coord = waypoints_data[name_point]["n_coord"]
407                     e_coord = waypoints_data[name_point]["e_coord"]
408                     alt = waypoints_data[name_point]["altitude"]
409                     speed = waypoints_data[name_point]["speed"]
410                     tf_leg(n_coord, e_coord, alt, speed)
411                     North_Coordinates2.append(n_coord)
412                     East_Coordinates2.append(e_coord)
413                     Altitudes2.append(alt)
414                     Speeds.append(speed)
415                     list = [n_coord, e_coord, alt, speed]
416                     check_position(list)
417                 if waypoints_data[name_point]["leg_type"] == "fp_Scan":
418                     ts = waypoints_data[name_point]["trackseparation"]
419                     f_p_n = waypoints_data[name_point]["point1_n"]
420                     f_p_e = waypoints_data[name_point]["point1_e"]
421                     v = waypoints_data[name_point]["speed"]
422                     h = waypoints_data[name_point]["altitude"]
423                     i = waypoints_data[name_point]["iteration"]
424                     scan_leg(ts, f_p_n, f_p_e, v, h, i)
425                 if waypoints_data[name_point]["leg_type"] == "fp_IterativeLeg":
426                     name_next_point = waypoints_data[name_point]["name_next_point"]
427                     num_of_iter = waypoints_data[name_point]["num_of_iter"]
428                     iterative_leg(name_point, num_of_iter, name_next_point)
429                 if waypoints_data[name_point]["leg_type"] == "fp_IntersectionLeg":
430                     limit = waypoints_data[name_point]["limit"]

```

```

431         option = input("Intersection Leg: to select one of the possibilities, insert a "
432             "number included between 1 and " + str(limit) + ": ")
433         option = int(option)
434         intersection_leg(name_point, option)
435         rip += 1
436         n_coord = waypoints_data[name_next_point]["n_coord"]
437         e_coord = waypoints_data[name_next_point]["e_coord"]
438         h = waypoints_data[name_next_point]["altitude"]
439         v = waypoints_data[name_next_point]["speed"]
440         tf_leg(n_coord, e_coord, h, v)
441         list1 = [n_coord, e_coord, h, v]
442         North_Coordinates2.append(list1[0])
443         East_Coordinates2.append(list1[1])
444         Altitudes2.append(list1[2])
445         Speeds.append(list1[3])
446         check_position(list1)
447
448
449     # Intersection_Leg Function
450
451     def intersection_leg(name_points, option):
452         name_point = waypoints_data[name_points]["name_point_possibility" + str(option-1)]
453         if waypoints_data[name_point]["leg_type"] == "fp_TFLeg":
454             n_coord = waypoints_data[name_point]["n_coord"]
455             e_coord = waypoints_data[name_point]["e_coord"]
456             alt = waypoints_data[name_point]["altitude"]
457             speed = waypoints_data[name_point]["speed"]
458             tf_leg(n_coord, e_coord, alt, speed)
459             North_Coordinates2.append(n_coord)
460             East_Coordinates2.append(e_coord)
461             Altitudes2.append(alt)
462             Speeds.append(speed)
463             list = [n_coord, e_coord, alt, speed]
464             check_position(list)
465         if waypoints_data[name_point]["leg_type"] == "fp_Scan":
466             ts = waypoints_data[name_point]["trackseparation"]
467             f_p_n = waypoints_data[name_point]["point1_n"]
468             f_p_e = waypoints_data[name_point]["point1_e"]
469             v = waypoints_data[name_point]["speed"]
470             h = waypoints_data[name_point]["altitude"]
471             i = waypoints_data[name_point]["iteration"]
472             scan_leg(ts, f_p_n, f_p_e, v, h, i)
473         if waypoints_data[name_point]["leg_type"] == "fp_IterativeLeg":
474             name_next_point = waypoints_data[name_point]["name_next_point"]

```



```

475         num_of_iter = waypoints_data[name_point]["num_of_iter"]
476         iterative_leg(name_point, num_of_iter, name_next_point)
477     if waypoints_data[name_point]["leg_type"] == "fp_IntersectionLeg":
478         limit = waypoints_data[name_point]["limit"]
479         option = input("Intersection Leg: to select one of the possibilities, insert a "
480                        "number included between 1 and " + str(limit) + ": ")
481         option = int(option)
482         intersection_leg(name_point, option)
483
484
485     # Leg Function
486
487     def leg(name_point):
488         if waypoints_data[name_point]["leg_type"] == "fp_TFLeg":
489             n_coord = waypoints_data[name_point]["n_coord"]
490             e_coord = waypoints_data[name_point]["e_coord"]
491             alt = waypoints_data[name_point]["altitude"]
492             speed = waypoints_data[name_point]["speed"]
493             tf_leg(n_coord, e_coord, alt, speed)
494             North_Coordinates2.append(n_coord)
495             East_Coordinates2.append(e_coord)
496             Altitudes2.append(alt)
497             Speeds.append(speed)
498             list = [n_coord, e_coord, alt, speed]
499             check_position(list)
500         if waypoints_data[name_point]["leg_type"] == "fp_Scan":
501             ts = waypoints_data[name_point]["trackseparation"]
502             f_p_n = waypoints_data[name_point]["point1_n"]
503             f_p_e = waypoints_data[name_point]["point1_e"]
504             v = waypoints_data[name_point]["speed"]
505             h = waypoints_data[name_point]["altitude"]
506             i = waypoints_data[name_point]["iteration"]
507             scan_leg(ts, f_p_n, f_p_e, v, h, i)
508         if waypoints_data[name_point]["leg_type"] == "fp_IterativeLeg":
509             name_next_point = waypoints_data[name_point]["name_next_point"]
510             num_of_iter = waypoints_data[name_point]["num_of_iter"]
511             iterative_leg(name_point, num_of_iter, name_next_point)
512         if waypoints_data[name_point]["leg_type"] == "fp_IntersectionLeg":
513             limit = waypoints_data[name_point]["limit"]
514             option = input("Intersection Leg: to select one of the possibilities, insert a "
515                           "number included between 1 and " + str(limit) + ": ")
516             option = int(option)
517             intersection_leg(name_point, option)
518

```

```

519
520 # GetFirstChild_of_a_Stage Function
521
522 def get_first_child_stage(name_stage):
523     legs = []
524     for stages in tree.iter(tag='stage'):
525         if stages.attrib['id'] == name_stage:
526             for components in stages.iter(tag='leg'):
527                 name_point = components.attrib['id']
528                 legs.append(name_point)
529     possibilities = []
530     for elem in tree.iter(tag='leg'):
531         if elem.attrib['xsi_type'] == "fp_IntersectionLeg":
532             for choices in elem.iter():
533                 possibilitie = choices.text
534                 possibilities = possibilitie.split()
535     next = []
536     iter_path = []
537     for elem in tree.iter(tag='leg'):
538         if elem.attrib['xsi_type'] == "fp_IterativeLeg":
539             for components in elem.iter():
540                 if components.tag == "body":
541                     iterative_path = components.text
542                     iter_path = iterative_path.split()
543                 elif components.tag == "next":
544                     name_next_point = components.text
545                     next.append(name_next_point)
546     matches1 = []
547     for leg1 in legs:
548         for possibility in possibilities:
549             if leg1 == possibility:
550                 matches1.append(leg1)
551     for n in range(len(matches1)):
552         legs.remove(matches1[n])
553     matches2 = []
554     for leg2 in legs:
555         for iter in iter_path:
556             if leg2 == iter:
557                 matches2.append(leg2)
558     for n in range(len(matches2)):
559         legs.remove(matches2[n])
560     matches3 = []
561     for leg3 in legs:
562         for nex in next:

```

```

563         if leg3 == nex:
564             matches3.append(leg3)
565         for n in range(len(matches3)):
566             legs.remove(matches3[n])
567         first_child = legs
568         return first_child
569
570
571     # Stage Function
572
573     def stage(name_stage):
574         name_points = get_first_child_stage(name_stage)
575         for n in range(len(name_points)):
576             leg(name_points[n])
577         t_stage = time.time() - t0
578         Time.append(t_stage)
579         sum = 0
580         for n in range(len(Time_For_Each_Leg)):
581             sum = sum + Time_For_Each_Leg[n]
582         Theoretical_Time_to_Plot.append(sum)
583         del Time_For_Each_Leg[:]
584
585
586     # ----- Reading and parsing XML File ----- #
587
588     FP = input("Which Flight Plan do you want to use? Please, select 0 for Lat/Long FP or 1 for NED FP: ")
589
590     if FP == '0':
591         tree = ET.parse('DroneFlightPlan_Neighborhood_Final_Version_LongLat.xml')
592     elif FP == '1':
593         tree = ET.parse('DroneFlightPlan_Neighborhood_Final_Version_meters.xml')
594
595     root = tree.getroot()
596     ET.tostring(root, encoding='utf8').decode('utf8')
597
598     LOG = input("To run the code with plots enter ON; to run the code without plots enter OFF: ")
599
600     # ----- Creation of an empty excel file to store points of real path ----- #
601
602     data_collection = xlswriter.Workbook('Data.xlsx')
603     worksheet = data_collection.add_worksheet()
604     worksheet.write(0, 0, "North Coordinates")
605     worksheet.write(1, 0, "East Coordinates")
606     worksheet.write(2, 0, "Altitudes")

```

```

607
608 # ----- Connection to AirSim ----- #
609
610 # Connection to AirSim simulator
611
612 client = airsim.MultirotorClient()
613 client.confirmConnection()
614 client.enableApiControl(True)
615 client.armDisarm(True)
616
617 state = client.getMultirotorState()
618 s = pprint.pformat(state)
619 print("state: %s" % s)
620
621 x_home = state.kinematics_estimated.position.x_val
622 y_home = state.kinematics_estimated.position.y_val
623 z_home = state.kinematics_estimated.position.z_val
624
625 lat0 = state.gps_location.latitude
626 long0 = state.gps_location.longitude
627 h_geo0 = state.gps_location.altitude
628
629 arcWE = geodesic.Geodesic.WGS84.Inverse(lat0, long0-0.5, lat0, long0+0.5)
630 arcNS = geodesic.Geodesic.WGS84.Inverse(lat0-0.5, long0, lat0+0.5, long0)
631
632 lat_conv = 1 / arcNS['s12']
633 long_conv = 1 / arcWE['s12']
634
635 # ----- Creation of a dictionary with all waypoints informations ----- #
636
637 waypoints_data = {}
638
639 if FP == '0':
640     for legs in tree.iter(tag='leg'):
641         if legs.attrib['xsi_type'] == "fp_TFLeg":
642             name_point = legs.attrib['id']
643             waypoints_data[name_point] = {}
644             leg_type = legs.attrib['xsi_type']
645             waypoints_data[name_point]["leg_type"] = leg_type
646             for components in legs.iter():
647                 if components.tag == "latitude":
648                     lat = components.text
649                     lat = float(lat)
650                     if lat == lat0:

```

```

651         n_coord = 0
652     else:
653         n_coord = (lat-lat0)/lat_conv
654         waypoints_data[name_point]["n_coord"] = n_coord
655     elif components.tag == "longitude":
656         long = components.text
657         long = float(long)
658         if long == long0:
659             e_coord = 0
660         else:
661             e_coord = (long-long0)/long_conv
662             waypoints_data[name_point]["e_coord"] = e_coord
663     elif components.tag == "altitude":
664         altitude = components.text
665         altitude = float(altitude)
666         alt = z_home - (altitude - h_geo0)
667         waypoints_data[name_point]["altitude"] = alt
668     elif components.tag == "speed":
669         speed = components.text
670         speed = float(speed)
671         waypoints_data[name_point]["speed"] = speed
672 if legs.attrib['xsi_type'] == "fp_Scan":
673     name_point = legs.attrib['id']
674     waypoints_data[name_point] = {}
675     leg_type = legs.attrib['xsi_type']
676     waypoints_data[name_point]["leg_type"] = leg_type
677     for components in legs.iter():
678         if components.tag == "trackseparation":
679             ts = components.text
680             ts = float(ts)
681             waypoints_data[name_point]["trackseparation"] = ts
682         elif components.tag == "point1":
683             f_p = components.text
684             f_p = f_p.split()
685             lat = float(f_p[0])
686             if lat == lat0:
687                 f_p_n = 0
688             else:
689                 f_p_n = (lat - lat0) / lat_conv
690             long = float(f_p[1])
691             if long == long0:
692                 f_p_e = 0
693             else:
694                 f_p_e = (long - long0) / long_conv

```

```

695         waypoints_data[name_point]["point1_n"] = f_p_n
696         waypoints_data[name_point]["point1_e"] = f_p_e
697     elif components.tag == "altitude":
698         h = components.text
699         h = float(h)
700         alt = z_home - (h - h_geo0)
701         waypoints_data[name_point]["altitude"] = alt
702     elif components.tag == "speed":
703         v = components.text
704         v = float(v)
705         waypoints_data[name_point]["speed"] = v
706     l_l = abs(f_p_n) + abs(f_p_e)
707     ll = round(l_l)
708     waypoints_data[name_point]["iteration"] = int(ll / ts)
709 if legs.attrib['xsi_type'] == "fp_IterativeLeg":
710     name_point = legs.attrib['id']
711     waypoints_data[name_point] = {}
712     leg_type = legs.attrib['xsi_type']
713     waypoints_data[name_point]["leg_type"] = leg_type
714     for components in legs.iter():
715         if components.tag == "body":
716             iterative_path = components.text
717             iter_path = iterative_path.split()
718             len_iter_path = len(iter_path)
719             waypoints_data[name_point]["len_iter_path"] = len_iter_path
720             for n in range(len(iter_path)):
721                 waypoints_data[name_point]["name_point_iter_path" + str(n)] = iter_path[n]
722         elif components.tag == "next":
723             name_next_point = components.text
724             waypoints_data[name_point]["name_next_point"] = name_next_point
725         elif components.tag == "upperBound":
726             num_of_iteration = components.text
727             num_of_iter = int(num_of_iteration)
728             waypoints_data[name_point]["num_of_iter"] = num_of_iter
729 if legs.attrib['xsi_type'] == "fp_IntersectionLeg":
730     name_point = legs.attrib['id']
731     waypoints_data[name_point] = {}
732     leg_type = legs.attrib['xsi_type']
733     waypoints_data[name_point]["leg_type"] = leg_type
734     for choices in legs.iter():
735         possibilities = choices.text
736         possibilities = possibilities.split()
737         for n in range(len(possibilities)):
738             waypoints_data[name_point]["name_point_possibility" + str(n)] = possibilities[n]

```



```

783         h = components.text
784         h = float(h)
785         waypoints_data[name_point]["altitude"] = h
786     elif components.tag == "speed":
787         v = components.text
788         v = float(v)
789         waypoints_data[name_point]["speed"] = v
790     l_l = abs(f_p_n) + abs(f_p_e)
791     ll = round(l_l)
792     waypoints_data[name_point]["iteration"] = int(ll / ts)
793 if legs.attrib['xsi_type'] == "fp_IterativeLeg":
794     name_point = legs.attrib['id']
795     waypoints_data[name_point] = {}
796     leg_type = legs.attrib['xsi_type']
797     waypoints_data[name_point]["leg_type"] = leg_type
798     for components in legs.iter():
799         if components.tag == "body":
800             iterative_path = components.text
801             iter_path = iterative_path.split()
802             len_iter_path = len(iter_path)
803             waypoints_data[name_point]["len_iter_path"] = len_iter_path
804             for n in range(len(iter_path)):
805                 waypoints_data[name_point]["name_point_iter_path" + str(n)] = iter_path[n]
806         elif components.tag == "next":
807             name_next_point = components.text
808             waypoints_data[name_point]["name_next_point"] = name_next_point
809         elif components.tag == "upperBound":
810             num_of_iteration = components.text
811             num_of_iter = int(num_of_iteration)
812             waypoints_data[name_point]["num_of_iter"] = num_of_iter
813 if legs.attrib['xsi_type'] == "fp_IntersectionLeg":
814     name_point = legs.attrib['id']
815     waypoints_data[name_point] = {}
816     leg_type = legs.attrib['xsi_type']
817     waypoints_data[name_point]["leg_type"] = leg_type
818     for choices in legs.iter():
819         possibilities = choices.text
820         possibilities = possibilities.split()
821         for n in range(len(possibilities)):
822             waypoints_data[name_point]["name_point_possibility" + str(n)] = possibilities[n]
823         limit = len(possibilities)
824         waypoints_data[name_point]["limit"] = limit
825
826 print(waypoints_data)

```



```

827
828 # ----- Take Off Part ----- #
829
830 client.takeoffAsync()
831 time.sleep(4)
832
833 t0 = time.time()
834
835 # ----- Execution of the Flight Plan ----- #
836
837 Time = []
838 North_Coordinates2 = [x_home]
839 East_Coordinates2 = [y_home]
840 Altitudes2 = [z_home]
841 Speeds = []
842 Time_For_Each_Leg = []
843 Theoretical_Time_to_Plot = []
844
845 column = 0
846
847 Stages = []
848 for element in root.iter(tag='stage'):
849     Stages.append(element.attrib['id'])
850
851 for n in range(len(Stages)):
852     stage(Stages[n])
853
854 # ----- Landing + Carlo Disarm ----- #
855
856 client.moveByVelocityZAsync(0, 0, z_home, 2, airsims.DrivetrainType.MaxDegreeOfFreedom, airsims.YawMode(False, 0)).join()
857
858 client.armDisarm(False)
859 client.enableApiControl(False)
860
861 data_collection.close()
862
863 # ----- Position Plots Part ----- #
864
865 # Real Path vs Theoretical Path Plot
866
867 if LOG == 'ON':
868     data = openpyxl.load_workbook('Data.xlsx')
869     sheet = data['Sheet1']
870

```

```
871     estremo_sx = []
872     estremo_dx = []
873     estremo_cn = []
874
875     all_columns = sheet.columns
876
877     for tupla in all_columns:
878         lista = list(tupla)
879         listino = [str(lista[0]), str(lista[1]), str(lista[2])]
880         sx = listino[0]
881         cn = listino[1]
882         dx = listino[2]
883         if len(dx) == 18 and len(sx) == 18 and len(cn) == 18:
884             e_dx = dx[15:17]
885             estremo_dx.append(e_dx)
886             e_cn = cn[15:17]
887             estremo_cn.append(e_cn)
888             e_sx = sx[15:17]
889             estremo_sx.append(e_sx)
890         elif len(dx) == 19 and len(sx) == 19 and len(cn) == 19:
891             e_dx = dx[15:18]
892             estremo_dx.append(e_dx)
893             e_cn = cn[15:18]
894             estremo_cn.append(e_cn)
895             e_sx = sx[15:18]
896             estremo_sx.append(e_sx)
897         elif len(dx) == 20 and len(sx) == 20 and len(cn) == 20:
898             e_dx = dx[15:19]
899             estremo_dx.append(e_dx)
900             e_cn = cn[15:19]
901             estremo_cn.append(e_cn)
902             e_sx = sx[15:19]
903             estremo_sx.append(e_sx)
904
905     estremo_sx.remove('A1')
906     estremo_cn.remove('A2')
907     estremo_dx.remove('A3')
908
909     Tutti_Punti = []
910
911     for n in range(len(estremo_sx)):
912         a = estremo_sx[n]
913         c = estremo_cn[n]
914         b = estremo_dx[n]
```

```

915         multiple_cells = sheet[a:b]
916         for row in multiple_cells:
917             for cell in row:
918                 list = cell.value
919                 Tutti_Punti.append(list)
920
921     North_Coordinates1 = []
922     East_Coordinates1 = []
923     Altitudes1 = []
924     l = len(Tutti_Punti)
925
926     for n in range(0, l, 3):
927         North_Coordinates1.append(Tutti_Punti[n])
928         East_Coordinates1.append(Tutti_Punti[n+1])
929         Altitudes1.append(-Tutti_Punti[n+2])
930
931     fig = plt.figure()
932     ax = plt.axes(projection='3d')
933     plt.gca().invert_yaxis()
934
935     Altitudes2_abs = []
936     for n in range(len(Altitudes2)):
937         alt = - (Altitudes2[n])
938         Altitudes2_abs.append(alt)
939
940     ax.plot3D(North_Coordinates1, East_Coordinates1, Altitudes1, color='blue')
941     ax.plot3D(North_Coordinates2, East_Coordinates2, Altitudes2_abs, color='red')
942     ax.set_xlabel('North Coordinate [m]')
943     ax.set_ylabel('East Coordinate [m]')
944     ax.set_zlabel('Absolute Value of Altitude [m]')
945
946
947     # Real and Theoretical Coordinates vs Time Plot
948
949     fig = plt.figure()
950     ax = plt.axes(projection='3d')
951
952     Tot = Time[len(Time)-1]
953
954     iteraz1 = len(North_Coordinates1)
955     iteraz2 = len(North_Coordinates2)
956     iteraz3 = len(East_Coordinates1)
957     iteraz4 = len(East_Coordinates2)
958     iteraz5 = len(Altitudes1)

```

```

959     iteraz6 = len(Altitudes2_abs)
960
961     x1 = np.linspace(0.0, Tot, iteraz1)
962     x2 = np.linspace(0.0, Tot, iteraz2)
963     x3 = np.linspace(0.0, Tot, iteraz3)
964     x4 = np.linspace(0.0, Tot, iteraz4)
965     x5 = np.linspace(0.0, Tot, iteraz5)
966     x6 = np.linspace(0.0, Tot, iteraz6)
967
968     y1 = North_Coordinates1
969     y2 = North_Coordinates2
970     y3 = East_Coordinates1
971     y4 = East_Coordinates2
972     y5 = Altitudes1
973     y6 = Altitudes2_abs
974
975     plt.subplot(3, 1, 1)
976     plt.plot(x1, y1, 'blue', x2, y2, 'red')
977
978     plt.subplot(3, 1, 2)
979     plt.plot(x3, y3, 'blue', x4, y4, 'red')
980
981     plt.subplot(3, 1, 3)
982     plt.plot(x5, y5, 'blue', x6, y6, 'red')
983     ax.grid()
984
985     # ----- Time Plots Part ----- #
986
987     Real_Time_to_Plot = []
988     Real_Time_to_Plot.append(Time[0])
989     for n in range(1, len(Time)):
990         Real_Time_to_Plot.append(Time[n]-Time[n-1])
991     Real_Time_to_Plot.append(Time[len(Time)-1])
992     print(Real_Time_to_Plot)
993
994     Total_Theoretical_Time = 0
995     for n in range(len(Theoretical_Time_to_Plot)):
996         Total_Theoretical_Time = Total_Theoretical_Time + Theoretical_Time_to_Plot[n]
997     Theoretical_Time_to_Plot.append(Total_Theoretical_Time)
998     print(Theoretical_Time_to_Plot)
999
1000     # First, Second, Third Stage Time Plot
1001
1002     for n in range(len(Real_Time_to_Plot)-1):

```

```

1003     fig = plt.figure()
1004     ax = plt.axes(projection="3d")
1005
1006     num_bars = 2
1007     x_pos = [1, 1]
1008     y_pos = [1, 5]
1009     z_pos = [0] * num_bars
1010     x_size = np.ones(num_bars)
1011     y_size = np.ones(num_bars)
1012     z_size = [Theoretical_Time_to_Plot[n], Real_Time_to_Plot[n]]
1013
1014     ax.bar3d(x_pos[0], y_pos[0], z_pos[0], x_size[0], y_size[0], z_size[0], color='blue')
1015     ax.bar3d(x_pos[1], y_pos[1], z_pos[1], x_size[1], y_size[1], z_size[1], color='red')
1016
1017     a = str(n+1)
1018
1019     ax.set_zlabel('Time spent [s]')
1020     ax.set_title(a + '° Stage Duration')
1021
1022 # Total Mission Time Plot
1023
1024 fig = plt.figure()
1025 ax = plt.axes(projection="3d")
1026
1027 l1 = len(Theoretical_Time_to_Plot)-1
1028 l2 = len(Real_Time_to_Plot)-1
1029
1030 num_bars = 2
1031 x_pos = [1, 1]
1032 y_pos = [1, 5]
1033 z_pos = [0] * num_bars
1034 x_size = np.ones(num_bars)
1035 y_size = np.ones(num_bars)
1036 z_size = [Theoretical_Time_to_Plot[l1], Real_Time_to_Plot[l2]]
1037
1038 ax.bar3d(x_pos[0], y_pos[0], z_pos[0], x_size[0], y_size[0], z_size[0], color='blue')
1039 ax.bar3d(x_pos[1], y_pos[1], z_pos[1], x_size[1], y_size[1], z_size[1], color='red')
1040
1041 ax.set_zlabel('Time spent [s]')
1042 ax.set_title(' Total Mission Duration')
1043
1044 plt.show()
1045
1046

```

- **MULTIPLE DRONES' FLIGHT PLAN: 1° DRONE (NED Coordinates)**

```

<FlightPlan>
  <Locale>
    <speedUnits>ms</speedUnits>
    <altitudeUnits>m</altitudeUnits>
    <trackseparationUnits>m</trackseparationUnits>
    <distanceUnits>m</distanceUnits>
    <decimalSeparator>.</decimalSeparator>
    <groupSeparator/>
  </Locale>

  <MainFP>
    <name>NEIGHBORHOOD FLIGHT PLAN.</name>
    <description>Neighborhood Surveillance Mission.</description>

    <stages>
      <stage id="First Part">
        <leg id="zero-point" xsi_type="fp_TFLeg">
          <dest>
            <name>N0</name>
            <north_coordinate>125</north_coordinate>
            <east_coordinate>0</east_coordinate>
            <altitude>-20</altitude>
            <speed>5</speed>
            <next>first-point</next>
          </dest>
        </leg>
      </stage>

      <stage id="Iterative Leg Part">
        <leg id="first-point" xsi_type="fp_IterativeLeg">
          <next>third-point</next>
          <body>second-point-one second-point-two second-point-three second-point-four</body>
          <upperBound>20</upperBound>
          <first>second-point-one</first>
          <last>second-point-four</last>
        </leg>
        <leg id="second-point-one" xsi_type="fp_TFLeg">
          <dest>
            <name>N21</name>
            <north_coordinate>125</north_coordinate>
            <east_coordinate>125</east_coordinate>
            <altitude>-30</altitude>
            <speed>10</speed>
            <next>second-point-two</next>
          </dest>
        </leg>
        <leg id="second-point-two" xsi_type="fp_TFLeg">

```

```

<leg id="second-point-two" xsi_type="fp_TFLeg">
  <dest>
    <name>N22</name>
    <north_coordinate>-125</north_coordinate>
    <east_coordinate>125</east_coordinate>
    <altitude>-30</altitude>
    <speed>10</speed>
    <next>second-point-three</next>
  </dest>
</leg>
<leg id="second-point-three" xsi_type="fp_TFLeg">
  <dest>
    <name>N23</name>
    <north_coordinate>-125</north_coordinate>
    <east_coordinate>-125</east_coordinate>
    <altitude>-30</altitude>
    <speed>10</speed>
    <next>second-point-four</next>
  </dest>
</leg>
<leg id="second-point-four" xsi_type="fp_TFLeg">
  <dest>
    <name>N24</name>
    <north_coordinate>125</north_coordinate>
    <east_coordinate>-125</east_coordinate>
    <altitude>-30</altitude>
    <speed>10</speed>
    <next>third-point</next>
  </dest>
</leg>
<leg id="third-point" xsi_type="fp_TFLeg">
  <dest>
    <name>N4</name>
    <north_coordinate>0</north_coordinate>
    <east_coordinate>0</east_coordinate>
    <altitude>-30</altitude>
    <speed>3</speed>
  </dest>
</leg>
</stage>
</stages>
</MainFP>
</FlightPlan>

```

- **MULTIPLE DRONES' FLIGHT PLAN: 1° DRONE (Geographical Coordinates)**

```

<FlightPlan>
  <Locale>
    <speedUnits>ms</speedUnits>
    <altitudeUnits>m</altitudeUnits>
    <trackseparationUnits>m</trackseparationUnits>
    <distanceUnits>°</distanceUnits>
    <decimalSeparator>.</decimalSeparator>
    <groupSeparator>/>
  </Locale>

  <MainFP>
    <name>NEIGHBORHOOD FLIGHT PLAN.</name>
    <description>Neighborhood Surveillance Mission.</description>

    <stages>
      <stage id="First Part">
        <leg id="zero-point" xsi_type="fp_TFLeg">
          <dest>
            <name>N0</name>
            <latitude>47.64260464285712</latitude>
            <longitude>-122.140365</longitude>
            <altitude>143.199297198</altitude>
            <speed>5</speed>
            <next>first-point</next>
          </dest>
        </leg>
      </stage>

      <stage id="Iterative Leg Part">
        <leg id="first-point" xsi_type="fp_IterativeLeg">
          <next>third-point</next>
          <body>second-point-one second-point-two second-point-three second-point-four</body>
          <upperBound>20</upperBound>
          <first>second-point-one</first>
          <last>second-point-four</last>
        </leg>
        <leg id="second-point-one" xsi_type="fp_TFLeg">
          <dest>
            <name>N21</name>
            <latitude>47.64260464285712</latitude>
            <longitude>-122.1386985308925</longitude>
            <altitude>153.199297198</altitude>
            <speed>10</speed>
            <next>second-point-two</next>
          </dest>
        </leg>
        <leg id="second-point-two" xsi_type="fp_TFLeg">

```



```

<leg id="second-point-two" xsi_type="fp_TFLeg">
  <dest>
    <name>N22</name>
    <latitude>47.640360097142874</latitude>
    <longitude>-122.1386985308925</longitude>
    <altitude>153.199297198</altitude>
    <speed>10</speed>
    <next>second-point-three</next>
  </dest>
</leg>
<leg id="second-point-three" xsi_type="fp_TFLeg">
  <dest>
    <name>N23</name>
    <latitude>47.640360097142874</latitude>
    <longitude>-122.14203146910751</longitude>
    <altitude>153.199297198</altitude>
    <speed>10</speed>
    <next>second-point-four</next>
  </dest>
</leg>
<leg id="second-point-four" xsi_type="fp_TFLeg">
  <dest>
    <name>N24</name>
    <latitude>47.64260464285712</latitude>
    <longitude>-122.14203146910751</longitude>
    <altitude>153.199297198</altitude>
    <speed>10</speed>
    <next>third-point</next>
  </dest>
</leg>
<leg id="third-point" xsi_type="fp_TFLeg">
  <dest>
    <name>N4</name>
    <latitude>47.64148237</latitude>
    <longitude>-122.140365</longitude>
    <altitude>153.199297198</altitude>
    <speed>3</speed>
  </dest>
</leg>
</stage>
</stages>
</MainFP>
</FlightPlan>

```

- **MULTIPLE DRONES' FLIGHT PLAN: 2° DRONE (NED Coordinates)**

```
<FlightPlan>
  <Locale>
    <speedUnits>ms</speedUnits>
    <altitudeUnits>m</altitudeUnits>
    <trackseparationUnits>m</trackseparationUnits>
    <distanceUnits>m</distanceUnits>
    <decimalSeparator>.</decimalSeparator>
    <groupSeparator/>
  </Locale>

  <MainFP>
    <name>NEIGHBORHOOD FLIGHT PLAN.</name>
    <description>Neighborhood Surveillance Mission.</description>

    <stages>
      <stage id="Intersection Leg Part">
        <leg id="zero-point" xsi_type="fp_IntersectionLeg">
          <nextList>third-point-a third-point-b third-point-c third-point-d</nextList>
        </leg>
        <leg id="third-point-a" xsi_type="fp_Scan">
          <dest>
            <coordinates>0 0</coordinates>
          </dest>
          <trackseparation>20</trackseparation>
          <area>
            <point1>125 0</point1>
            <point2>125 125</point2>
            <point3>0 125</point3>
            <point4>0 0</point4>
          </area>
          <speed>5</speed>
          <altitude>-40</altitude>
        </leg>
        <leg id="third-point-b" xsi_type="fp_Scan">
          <dest>
            <coordinates>0 0</coordinates>
          </dest>
          <trackseparation>20</trackseparation>
          <area>
            <point1>-125 0</point1>
            <point2>-125 -125</point2>
            <point3>0 -125</point3>
            <point4>0 0</point4>
          </area>
          <speed>5</speed>
          <altitude>-40</altitude>
        </leg>
        <leg id="third-point-c" xsi_type="fp_Scan">
```

```

<leg id="third-point-c" xsi_type="fp_Scan">
  <dest>
    <coordinates>0 0</coordinates>
  </dest>
  <trackseparation>20</trackseparation>
  <area>
    <point1>0 -125</point1>
    <point2>125 -125</point2>
    <point3>125 0</point3>
    <point4>0 0</point4>
  </area>
  <speed>5</speed>
  <altitude>-40</altitude>
</leg>
<leg id="third-point-d" xsi_type="fp_Scan">
  <dest>
    <coordinates>0 0</coordinates>
  </dest>
  <trackseparation>20</trackseparation>
  <area>
    <point1>0 125</point1>
    <point2>-125 125</point2>
    <point3>-125 0</point3>
    <point4>0 0</point4>
  </area>
  <speed>5</speed>
  <altitude>-40</altitude>
</leg>
<initialLegs> zero-point </initialLegs>
<finalLegs> third-point-a third-point-b third-point-c third-point-d </finalLegs>
</stage>

<stage id="Go Home Part">
  <leg id="Home1" xsi_type="fp_TFLeg">
    <dest>
      <name>Home1</name>
      <north_coordinate>0</north_coordinate>
      <east_coordinate>0</east_coordinate>
      <altitude>-4</altitude>
      <speed>3</speed>
      <next>Home2</next>
    </dest>
  </leg>
</stage>

</stages>

</MainFP>

</FlightPlan>

```

- **MULTIPLE DRONES' FLIGHT PLAN: 2° DRONE (Geographical Coordinates)**

```

<FlightPlan>
  <Locale>
    <speedUnits>ms</speedUnits>
    <altitudeUnits>m</altitudeUnits>
    <trackseparationUnits>m</trackseparationUnits>
    <distanceUnits>°</distanceUnits>
    <decimalSeparator>.</decimalSeparator>
    <groupSeparator/>
  </Locale>

  <MainFP>
    <name>NEIGHBORHOOD FLIGHT PLAN.</name>
    <description>Neighborhood Surveillance Mission.</description>

    <stages>
      <stage id="Intersection Leg Part">
        <leg id="zero-point" xsi_type="fp_IntersectionLeg">
          <nextList>third-point-a third-point-b third-point-c third-point-d</nextList>
        </leg>
        <leg id="third-point-a" xsi_type="fp_Scan">
          <dest>
            <coordinates>0 0</coordinates>
          </dest>
          <trackseparation>20</trackseparation>
          <area>
            <point1>47.64260464285712 -122.140365</point1>
            <point2>47.640360097142874 -122.1386985308925</point2>
            <point3>47.64148237 -122.1386985308925</point3>
            <point4>47.64148237 -122.140365</point4>
          </area>
          <speed>5</speed>
          <altitude>163.199297198</altitude>
        </leg>
        <leg id="third-point-b" xsi_type="fp_Scan">
          <dest>
            <coordinates>0 0</coordinates>
          </dest>
          <trackseparation>20</trackseparation>
          <area>
            <point1>47.640360097142874 -122.140365</point1>
            <point2>47.640360097142874 -122.14203146910751</point2>
            <point3>47.64148237 -122.14203146910751</point3>
            <point4>47.64148237 -122.140365</point4>
          </area>
          <speed>5</speed>
          <altitude>163.199297198</altitude>
        </leg>
        <leg id="third-point-c" xsi_type="fp_Scan">

```

```

<leg id="third-point-c" xsi_type="fp_Scan">
  <dest>
    <coordinates>0 0</coordinates>
  </dest>
  <trackseparation>20</trackseparation>
  <area>
    <point1>47.64148237 -122.14203146910751</point1>
    <point2>47.64260464285712 -122.14203146910751</point2>
    <point3>47.64260464285712 -122.140365</point3>
    <point4>47.64148237 -122.140365</point4>
  </area>
  <speed>5</speed>
  <altitude>163.199297198</altitude>
</leg>
<leg id="third-point-d" xsi_type="fp_Scan">
  <dest>
    <coordinates>0 0</coordinates>
  </dest>
  <trackseparation>20</trackseparation>
  <area>
    <point1>47.64148237 -122.1386985308925</point1>
    <point2>47.640360097142874 -122.1386985308925</point2>
    <point3>47.640360097142874 -122.140365</point3>
    <point4>47.64148237 -122.140365</point4>
  </area>
  <speed>5</speed>
  <altitude>163.199297198</altitude>
</leg>
<initialLegs> zero-point </initialLegs>
<finalLegs> third-point-a third-point-b third-point-c third-point-d </finalLegs>
</stage>

<stage id="Go Home Part">
  <leg id="Home1" xsi_type="fp_TFLeg">
    <dest>
      <name>Home1</name>
      <latitude>47.64148237</latitude>
      <longitude>-122.140365</longitude>
      <altitude>127.199297198</altitude>
      <speed>3</speed>
      <next>Home2</next>
    </dest>
  </leg>
</stage>

</stages>

</MainFP>

</FlightPlan>

```

- **MULTIPLE DRONES' PYTHON CODE**

```
1  # Author of the Code: Francesco Rose
2  # Master Thesis Project, Universitat Politècnica de Catalunya
3  # Dynamic Interface AirSim/XML Fight Plan
4
5  import airsims
6
7  import threading
8  import time
9  import concurrent.futures
10
11  from math import *
12
13  import xml.etree.ElementTree as ET
14
15  import numpy as np
16
17  from geographiclib import geodesic
18
19
20  # -----
21
22  client = airsims.MultirotorClient()
23  client.confirmConnection()
24
25  client.reset()
26
27  populations = input("How many drone do you want to use?: ")
28  population = int(populations)
29
30  for count in range(population):
31      (client.enableApiControl(True, vehicle_name='Drone' + str(count + 1)))
32      (client.armDisarm(True, vehicle_name='Drone' + str(count + 1)))
33
34      i = 1
35      h = -1
36
37  for count in range(population):
38      client.moveToZAsync(h-i, 3, vehicle_name='Drone' + str(count + 1))
39      i += 3
40      time.sleep(4)
41
42
```

```

43 class MainProgram:
44     def __init__(self):
45         self._lock = threading.Lock()
46
47     def execution(self, drone_number):
48
49         self._lock.acquire()
50
51         FP = input("Which Flight Plan do you want to use? Please, select 0 for Geographical FP or 1 for NED FP: ")
52
53         if FP == '0':
54             tree = ET.parse(
55                 'Drone' + str(drone_number + 1) + '_FlightPlan_Neighborhood_Final_Version_Geographical.xml')
56         elif FP == '1':
57             tree = ET.parse('Drone' + str(drone_number + 1) + '_FlightPlan_Neighborhood_Final_Version_meters.xml')
58
59         root = tree.getroot()
60         ET.tostring(root, encoding='utf8').decode('utf8')
61
62         vehicle_name = 'Drone' + str(drone_number + 1)
63
64         state = client.getMultirotorState(vehicle_name)
65
66         global x_home, y_home, z_home
67
68         x_home = state.kinematics_estimated.position.x_val
69         y_home = state.kinematics_estimated.position.y_val
70         z_home = state.kinematics_estimated.position.z_val
71
72         lat0 = state.gps_location.latitude
73         long0 = state.gps_location.longitude
74         h_geo0 = state.gps_location.altitude
75
76         arcWE = geodesic.Geodesic.WGS84.Inverse(lat0, long0 - 0.5, lat0, long0 + 0.5)
77         arcNS = geodesic.Geodesic.WGS84.Inverse(lat0 - 0.5, long0, lat0 + 0.5, long0)
78
79         lat_conv = 1 / arcNS['s12']
80         long_conv = 1 / arcWE['s12']
81
82         waypoints_data = {}
83
84         if FP == '0':
85             for legs in tree.iter(tag='leg'):
86                 if legs.attrib['xsi_type'] == "fp_TFLeg":

```

```

87     name_point = legs.attrib['id']
88     waypoints_data[name_point] = {}
89     leg_type = legs.attrib['xsi_type']
90     waypoints_data[name_point]["leg_type"] = leg_type
91     for components in legs.iter():
92         if components.tag == "latitude":
93             lat = components.text
94             lat = float(lat)
95             if lat == lat0:
96                 n_coord = 0
97             else:
98                 n_coord = (lat - lat0) / lat_conv
99             waypoints_data[name_point]["n_coord"] = n_coord
100         elif components.tag == "longitude":
101             long = components.text
102             long = float(long)
103             if long == long0:
104                 e_coord = 0
105             else:
106                 e_coord = (long - long0) / long_conv
107             waypoints_data[name_point]["e_coord"] = e_coord
108         elif components.tag == "altitude":
109             altitude = components.text
110             altitude = float(altitude)
111             alt = z_home - (altitude - h_geo0)
112             waypoints_data[name_point]["altitude"] = alt
113         elif components.tag == "speed":
114             speed = components.text
115             speed = float(speed)
116             waypoints_data[name_point]["speed"] = speed
117     if legs.attrib['xsi_type'] == "fp_Scan":
118         name_point = legs.attrib['id']
119         waypoints_data[name_point] = {}
120         leg_type = legs.attrib['xsi_type']
121         waypoints_data[name_point]["leg_type"] = leg_type
122         for components in legs.iter():
123             if components.tag == "trackseparation":
124                 ts = components.text
125                 ts = float(ts)
126                 waypoints_data[name_point]["trackseparation"] = ts
127             elif components.tag == "point1":
128                 f_p = components.text
129                 f_p = f_p.split()
130                 lat = float(f_p[0])

```



```

131         if lat == lat0:
132             f_p_n = 0
133         else:
134             f_p_n = (lat - lat0) / lat_conv
135         long = float(f_p[1])
136         if long == long0:
137             f_p_e = 0
138         else:
139             f_p_e = (long - long0) / long_conv
140         waypoints_data[name_point]["point1_n"] = f_p_n
141         waypoints_data[name_point]["point1_e"] = f_p_e
142     elif components.tag == "altitude":
143         h = components.text
144         h = float(h)
145         alt = z_home - (h - h_geo0)
146         waypoints_data[name_point]["altitude"] = alt
147     elif components.tag == "speed":
148         v = components.text
149         v = float(v)
150         waypoints_data[name_point]["speed"] = v
151     l_l = abs(f_p_n) + abs(f_p_e)
152     ll = round(l_l)
153     waypoints_data[name_point]["length_side"] = ll
154     waypoints_data[name_point]["iteration"] = int(ll / ts)
155     if legs.attrib['xsi_type'] == "fp_IterativeLeg":
156         name_point = legs.attrib['id']
157         waypoints_data[name_point] = {}
158         leg_type = legs.attrib['xsi_type']
159         waypoints_data[name_point]["leg_type"] = leg_type
160         for components in legs.iter():
161             if components.tag == "body":
162                 iterative_path = components.text
163                 iter_path = iterative_path.split()
164                 len_iter_path = len(iter_path)
165                 waypoints_data[name_point]["len_iter_path"] = len_iter_path
166                 for n in range(len(iter_path)):
167                     waypoints_data[name_point]["name_point_iter_path" + str(n)] = iter_path[n]
168             elif components.tag == "next":
169                 name_next_point = components.text
170                 waypoints_data[name_point]["name_next_point"] = name_next_point
171             elif components.tag == "upperBound":
172                 num_of_iteration = components.text
173                 num_of_iter = int(num_of_iteration)
174                 waypoints_data[name_point]["num_of_iter"] = num_of_iter

```

```

175         if legs.attrib['xsi_type'] == "fp_IntersectionLeg":
176             name_point = legs.attrib['id']
177             waypoints_data[name_point] = {}
178             leg_type = legs.attrib['xsi_type']
179             waypoints_data[name_point]["leg_type"] = leg_type
180             for choices in legs.iter():
181                 possibilities = choices.text
182                 possibilities = possibilities.split()
183                 for n in range(len(possibilities)):
184                     waypoints_data[name_point]["name_point_possibility" + str(n)] = possibilities[n]
185                 limit = len(possibilities)
186                 waypoints_data[name_point]["limit"] = limit
187     elif FP == '1':
188         for legs in tree.iter(tag='leg'):
189             if legs.attrib['xsi_type'] == "fp_TFLeg":
190                 name_point = legs.attrib['id']
191                 waypoints_data[name_point] = {}
192                 leg_type = legs.attrib['xsi_type']
193                 waypoints_data[name_point]["leg_type"] = leg_type
194                 for components in legs.iter():
195                     if components.tag == "north_coordinate":
196                         n_c = components.text
197                         n_coord = float(n_c)
198                         waypoints_data[name_point]["n_coord"] = n_coord
199                     elif components.tag == "east_coordinate":
200                         e_c = components.text
201                         e_coord = float(e_c)
202                         waypoints_data[name_point]["e_coord"] = e_coord
203                     elif components.tag == "altitude":
204                         alt = components.text
205                         alt = float(alt)
206                         waypoints_data[name_point]["altitude"] = alt
207                     elif components.tag == "speed":
208                         speed = components.text
209                         speed = float(speed)
210                         waypoints_data[name_point]["speed"] = speed
211             if legs.attrib['xsi_type'] == "fp_Scan":
212                 name_point = legs.attrib['id']
213                 waypoints_data[name_point] = {}
214                 leg_type = legs.attrib['xsi_type']
215                 waypoints_data[name_point]["leg_type"] = leg_type
216                 for components in legs.iter():
217                     if components.tag == "trackseparation":
218                         ts = components.text

```

```

219         ts = float(ts)
220         waypoints_data[name_point]["trackseparation"] = ts
221     elif components.tag == "point1":
222         f_p = components.text
223         f_p = f_p.split()
224         f_p_n = float(f_p[0])
225         f_p_e = float(f_p[1])
226         waypoints_data[name_point]["point1_n"] = f_p_n
227         waypoints_data[name_point]["point1_e"] = f_p_e
228     elif components.tag == "altitude":
229         h = components.text
230         h = float(h)
231         waypoints_data[name_point]["altitude"] = h
232     elif components.tag == "speed":
233         v = components.text
234         v = float(v)
235         waypoints_data[name_point]["speed"] = v
236     l_l = abs(f_p_n) + abs(f_p_e)
237     ll = round(l_l)
238     waypoints_data[name_point]["length_side"] = ll
239     waypoints_data[name_point]["iteration"] = int(ll / ts)
240     if legs.attrib['xsi_type'] == "fp_IterativeLeg":
241         name_point = legs.attrib['id']
242         waypoints_data[name_point] = {}
243         leg_type = legs.attrib['xsi_type']
244         waypoints_data[name_point]["leg_type"] = leg_type
245         for components in legs.iter():
246             if components.tag == "body":
247                 iterative_path = components.text
248                 iter_path = iterative_path.split()
249                 len_iter_path = len(iter_path)
250                 waypoints_data[name_point]["len_iter_path"] = len_iter_path
251                 for n in range(len(iter_path)):
252                     waypoints_data[name_point]["name_point_iter_path" + str(n)] = iter_path[n]
253             elif components.tag == "next":
254                 name_next_point = components.text
255                 waypoints_data[name_point]["name_next_point"] = name_next_point
256             elif components.tag == "upperBound":
257                 num_of_iteration = components.text
258                 num_of_iter = int(num_of_iteration)
259                 waypoints_data[name_point]["num_of_iter"] = num_of_iter
260     if legs.attrib['xsi_type'] == "fp_IntersectionLeg":
261         name_point = legs.attrib['id']
262         waypoints_data[name_point] = {}

```

```

263         leg_type = legs.attrib['xsi_type']
264         waypoints_data[name_point]['leg_type'] = leg_type
265         for choices in legs.iter():
266             possibilities = choices.text
267             possibilities = possibilities.split()
268             for n in range(len(possibilities)):
269                 waypoints_data[name_point]['name_point_possibility' + str(n)] = possibilities[n]
270             limit = len(possibilities)
271             waypoints_data[name_point]['limit'] = limit
272
273     print(waypoints_data)
274
275     self._lock.release()
276
277     stages = []
278     for element in root.iter(tag='stage'):
279         stages.append(element.attrib['id'])
280
281     legs = []
282     for n in range(len(stages)):
283         for stage in tree.iter(tag='stage'):
284             if stage.attrib['id'] == stages[n]:
285                 for components in stage.iter(tag='leg'):
286                     name_point = components.attrib['id']
287                     legs.append(name_point)
288
289     possibilities = []
290     for elem in tree.iter(tag='leg'):
291         if elem.attrib['xsi_type'] == "fp_IntersectionLeg":
292             for choices in elem.iter():
293                 pox = choices.text
294                 possibilities = pox.split()
295
296     iter_path = []
297     going = []
298     for elem in tree.iter(tag='leg'):
299         if elem.attrib['xsi_type'] == "fp_IterativeLeg":
300             for components in elem.iter():
301                 if components.tag == "body":
302                     iterative_path = components.text
303                     iter_path = iterative_path.split()
304                 elif components.tag == "next":
305                     name_next_point = components.text
306                     going.append(name_next_point)

```

```

307         for possibility in possibilities:
308             if leg1 == possibility:
309                 matches1.append(leg1)
310         for n in range(len(matches1)):
311             legs.remove(matches1[n])
312         matches2 = []
313         for leg2 in legs:
314             for iter in iter_path:
315                 if leg2 == iter:
316                     matches2.append(leg2)
317         for n in range(len(matches2)):
318             legs.remove(matches2[n])
319         matches3 = []
320         for leg3 in legs:
321             for nex in going:
322                 if leg3 == nex:
323                     matches3.append(leg3)
324         for n in range(len(matches3)):
325             legs.remove(matches3[n])
326         first_child = legs
327
328         for n in range(len(first_child)):
329             if waypoints_data[first_child[n]]["leg_type"] == "fp_TFLeg":
330                 n_coord = waypoints_data[first_child[n]]["n_coord"]
331                 e_coord = waypoints_data[first_child[n]]["e_coord"]
332                 alt = waypoints_data[first_child[n]]["altitude"]
333                 speed = waypoints_data[first_child[n]]["speed"]
334
335                 client.moveToPositionAsync(n_coord, e_coord, alt, speed, vehicle_name=vehicle_name)
336
337                 list1 = [n_coord, e_coord, alt, speed]
338                 state = client.getMultirotorState(vehicle_name)
339                 p_attuale = [state.kinematics_estimated.position.x_val, state.kinematics_estimated.position.y_val,
340                             state.kinematics_estimated.position.z_val]
341
342                 distance = sqrt(
343                     (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
344                         list1[2] - p_attuale[2]) ** 2)
345                 time_required = (distance / list1[3])
346
347                 time.sleep(time_required * 0.9)
348                 a = True
349                 while a:
350                     if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < p_attuale[1] < (

```

```

351         list1[1] + 2) and (list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
352             a = False
353         else:
354             time.sleep(0.1)
355
356         state = client.getMultirotorState(vehicle_name)
357         p_attuale = [state.kinematics_estimated.position.x_val,
358                     state.kinematics_estimated.position.y_val,
359                     state.kinematics_estimated.position.z_val]
360     if waypoints_data[first_child[n]]["leg_type"] == "fp_Scan":
361         ts = waypoints_data[first_child[n]]["trackseparation"]
362         f_p_n = waypoints_data[first_child[n]]["point1_n"]
363         f_p_e = waypoints_data[first_child[n]]["point1_e"]
364         v = waypoints_data[first_child[n]]["speed"]
365         h = waypoints_data[first_child[n]]["altitude"]
366         i = waypoints_data[first_child[n]]["iteration"]
367         ll = waypoints_data[first_child[n]]["length_side"]
368
369         client.moveToPositionAsync(f_p_n, f_p_e, h, v, vehicle_name=vehicle_name)
370
371         list1 = [f_p_n, f_p_e, h, v]
372         state = client.getMultirotorState(vehicle_name)
373         p_attuale = [state.kinematics_estimated.position.x_val, state.kinematics_estimated.position.y_val,
374                     state.kinematics_estimated.position.z_val]
375
376         distance = sqrt((list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
377             list1[2] - p_attuale[2]) ** 2)
378         time_required = (distance / list1[3])
379
380         time.sleep(time_required * 0.9)
381         a = True
382         while a:
383             if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < p_attuale[1] < (
384                 list1[1] + 2) and (list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
385                 a = False
386             else:
387                 time.sleep(0.1)
388
389             state = client.getMultirotorState(vehicle_name)
390             p_attuale = [state.kinematics_estimated.position.x_val,
391                         state.kinematics_estimated.position.y_val,
392                         state.kinematics_estimated.position.z_val]
393         if (f_p_e < 0 < f_p_n) or (f_p_n > 0 and f_p_e == 0):
394             if i % 2 == 0:

```

```

395 p = []
396 s = []
397 t = []
398 q = []
399 for num in range(1, i, 2):
400     sott1 = num * ts
401     a = f_p_n - sott1
402     P = (a, f_p_e, h, v)
403     p.append(P)
404     S = (a, f_p_e + ll, h, v)
405     s.append(S)
406 for mol in range(2, i + 1, 2):
407     sott2 = mol * ts
408     b = f_p_n - sott2
409     T = (b, f_p_e + ll, h, v)
410     t.append(T)
411     Q = (b, f_p_e, h, v)
412     q.append(Q)
413 Total = []
414 for n in range(len(q)):
415     total = [p[n], s[n], t[n], q[n]]
416     Total.append(total)
417 Totale = []
418 for n in range(len(Total)):
419     for a in range(4):
420         totale = Total[n][a]
421         Totale.append(totale)
422 print(Totale)
423 for n in range(len(Totale)):
424     client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3],
425                               vehicle_name=vehicle_name)
426
427 list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
428 state = client.getMultirotorState(vehicle_name)
429 p_attuale = [state.kinematics_estimated.position.x_val,
430              state.kinematics_estimated.position.y_val,
431              state.kinematics_estimated.position.z_val]
432
433 distance = sqrt(
434     (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
435         list1[2] - p_attuale[2]) ** 2)
436 time_required = (distance / list1[3])
437
438 time.sleep(time_required * 0.9)

```

```

439         a = True
440         while a:
441             if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < p_attuale[
442                 1] < (
443                     list1[1] + 2) and (
444                         list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
445                 a = False
446             else:
447                 time.sleep(0.1)
448
449             state = client.getMultirotorState(vehicle_name)
450             p_attuale = [state.kinematics_estimated.position.x_val,
451                         state.kinematics_estimated.position.y_val,
452                         state.kinematics_estimated.position.z_val]
453
454         else:
455             p = []
456             s = []
457             t = []
458             q = []
459             for num in range(1, i + 1, 2):
460                 sott1 = num * ts
461                 a = f_p_n - sott1
462                 P = (a, f_p_e, h, v)
463                 p.append(P)
464                 S = (a, f_p_e + ll, h, v)
465                 s.append(S)
466             for mol in range(2, i + 2, 2):
467                 sott2 = mol * ts
468                 b = f_p_n - sott2
469                 T = (b, f_p_e + ll, h, v)
470                 t.append(T)
471                 Q = (b, f_p_e, h, v)
472                 q.append(Q)
473             Total = []
474             for n in range(len(s)):
475                 total = [p[n], s[n], t[n], q[n]]
476                 Total.append(total)
477             Totale = []
478             for n in range(len(Total)):
479                 for a in range(4):
480                     totale = Total[n][a]
481                     Totale.append(totale)
482             print(Totale)
483             limit = len(Totale) - 2

```



```

483     for n in range(limit):
484         client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3],
485                                   vehicle_name=vehicle_name)
486
487         list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
488         state = client.getMultirotorState(vehicle_name)
489         p_attuale = [state.kinematics_estimated.position.x_val,
490                     state.kinematics_estimated.position.y_val,
491                     state.kinematics_estimated.position.z_val]
492
493         distance = sqrt(
494             (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
495                 list1[2] - p_attuale[2]) ** 2)
496         time_required = (distance / list1[3])
497
498         time.sleep(time_required * 0.9)
499         a = True
500         while a:
501             if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < p_attuale[
502                 1] < (
503                     list1[1] + 2) and (
504                         list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
505                 a = False
506             else:
507                 time.sleep(0.1)
508
509             state = client.getMultirotorState(vehicle_name)
510             p_attuale = [state.kinematics_estimated.position.x_val,
511                         state.kinematics_estimated.position.y_val,
512                         state.kinematics_estimated.position.z_val]
513         elif (f_p_n > 0 and f_p_e > 0) or (f_p_n == 0 and f_p_e > 0):
514             if i % 2 == 0:
515                 p = []
516                 s = []
517                 t = []
518                 q = []
519                 for num in range(1, i, 2):
520                     sott1 = num * ts
521                     a = f_p_n - sott1
522                     P = (a, f_p_e, h, v)
523                     p.append(P)
524                     S = (a, f_p_e - ll, h, v)
525                     s.append(S)
526                 for mol in range(2, i + 1, 2):

```

```

527     sott2 = mol * ts
528     b = f_p_n - sott2
529     T = (b, f_p_e - ll, h, v)
530     t.append(T)
531     Q = (b, f_p_e, h, v)
532     q.append(Q)
533     Total = []
534     for n in range(len(q)):
535         total = [p[n], s[n], t[n], q[n]]
536         Total.append(total)
537     Totale = []
538     for n in range(len(Total)):
539         for a in range(4):
540             totale = Total[n][a]
541             Totale.append(totale)
542     print(Totale)
543     for n in range(len(Totale)):
544         client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3],
545                                   vehicle_name=vehicle_name)
546
547     list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
548     state = client.getMultirotorState(vehicle_name)
549     p_attuale = [state.kinematics_estimated.position.x_val,
550                 state.kinematics_estimated.position.y_val,
551                 state.kinematics_estimated.position.z_val]
552
553     distance = sqrt(
554         (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
555             list1[2] - p_attuale[2]) ** 2)
556     time_required = (distance / list1[3])
557
558     time.sleep(time_required * 0.9)
559     a = True
560     while a:
561         if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < p_attuale[
562             1] < (
563                 list1[1] + 2) and (
564                     list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
565             a = False
566         else:
567             time.sleep(0.1)
568
569         state = client.getMultirotorState(vehicle_name)
570         p_attuale = [state.kinematics_estimated.position.x_val,

```

```

571 state.kinematics_estimated.position.y_val,
572 state.kinematics_estimated.position.z_val]
573
574 else:
575     p = []
576     s = []
577     t = []
578     q = []
579     for num in range(1, i + 1, 2):
580         sott1 = num * ts
581         a = f_p_n - sott1
582         P = (a, f_p_e, h, v)
583         p.append(P)
584         S = (a, f_p_e - ll, h, v)
585         s.append(S)
586     for mol in range(2, i + 2, 2):
587         sott2 = mol * ts
588         b = f_p_n - sott2
589         T = (b, f_p_e - ll, h, v)
590         t.append(T)
591         Q = (b, f_p_e, h, v)
592         q.append(Q)
593     Total = []
594     for n in range(len(s)):
595         total = [p[n], s[n], t[n], q[n]]
596         Total.append(total)
597     Totale = []
598     for n in range(len(Total)):
599         for a in range(4):
600             totale = Total[n][a]
601             Totale.append(totale)
602     print(Totale)
603     limit = len(Totale) - 2
604     for n in range(limit):
605         client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3],
606                                   vehicle_name=vehicle_name)
607
608     list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
609     state = client.getMultirotorState(vehicle_name)
610     p_attuale = [state.kinematics_estimated.position.x_val,
611                  state.kinematics_estimated.position.y_val,
612                  state.kinematics_estimated.position.z_val]
613
614     distance = sqrt(
615         (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (

```

```

615         list1[2] - p_attuale[2]) ** 2)
616         time_required = (distance / list1[3])
617
618         time.sleep(time_required * 0.9)
619         a = True
620         while a:
621             if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < p_attuale[
622                 1] < (
623                     list1[1] + 2) and (
624                         list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
625                 a = False
626             else:
627                 time.sleep(0.1)
628
629                 state = client.getMultirotorState(vehicle_name)
630                 p_attuale = [state.kinematics_estimated.position.x_val,
631                     state.kinematics_estimated.position.y_val,
632                     state.kinematics_estimated.position.z_val]
633         elif (f_p_n < 0 < f_p_e) or (f_p_n < 0 and f_p_e == 0):
634             if i % 2 == 0:
635                 p = []
636                 s = []
637                 t = []
638                 q = []
639                 for num in range(1, i, 2):
640                     sott1 = num * ts
641                     a = f_p_n + sott1
642                     p = (a, f_p_e, h, v)
643                     p.append(p)
644                     S = (a, f_p_e - ll, h, v)
645                     s.append(S)
646                 for mol in range(2, i + 1, 2):
647                     sott2 = mol * ts
648                     b = f_p_n + sott2
649                     T = (b, f_p_e - ll, h, v)
650                     t.append(T)
651                     Q = (b, f_p_e, h, v)
652                     q.append(Q)
653                 Total = []
654                 for n in range(len(q)):
655                     total = [p[n], s[n], t[n], q[n]]
656                     Total.append(total)
657                 Totale = []
658                 for n in range(len(Total)):

```

```

659         for a in range(4):
660             totale = Total[n][a]
661             Totale.append(totale)
662         print(Totale)
663         for n in range(len(Totale)):
664             client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3],
665                                       vehicle_name=vehicle_name)
666
667             list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
668             state = client.getMultirotorState(vehicle_name)
669             p_attuale = [state.kinematics_estimated.position.x_val,
670                           state.kinematics_estimated.position.y_val,
671                           state.kinematics_estimated.position.z_val]
672
673             distance = sqrt(
674                 (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
675                     list1[2] - p_attuale[2]) ** 2)
676             time_required = (distance / list1[3])
677
678             time.sleep(time_required * 0.9)
679             a = True
680             while a:
681                 if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < p_attuale[
682                     1] < (
683                         list1[1] + 2) and (
684                             list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
685                     a = False
686                 else:
687                     time.sleep(0.1)
688
689                 state = client.getMultirotorState(vehicle_name)
690                 p_attuale = [state.kinematics_estimated.position.x_val,
691                               state.kinematics_estimated.position.y_val,
692                               state.kinematics_estimated.position.z_val]
693             else:
694                 p = []
695                 s = []
696                 t = []
697                 q = []
698                 for num in range(1, i + 1, 2):
699                     sott1 = num * ts
700                     a = f_p_n + sott1
701                     p = (a, f_p_e, h, v)
702                     p.append(P)

```

```

703     S = (a, f_p_e - ll, h, v)
704     s.append(S)
705     for mol in range(2, i + 2, 2):
706         sott2 = mol * ts
707         b = f_p_n + sott2
708         T = (b, f_p_e - ll, h, v)
709         t.append(T)
710         Q = (b, f_p_e, h, v)
711         q.append(Q)
712     Total = []
713     for n in range(len(s)):
714         total = [p[n], s[n], t[n], q[n]]
715         Total.append(total)
716     Totale = []
717     for n in range(len(Total)):
718         for a in range(4):
719             totale = Total[n][a]
720             Totale.append(totale)
721     limit = len(Totale) - 2
722     for n in range(limit):
723         client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3],
724                                   vehicle_name=vehicle_name)
725
726         list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
727         state = client.getMultirotorState(vehicle_name)
728         p_attuale = [state.kinematics_estimated.position.x_val,
729                     state.kinematics_estimated.position.y_val,
730                     state.kinematics_estimated.position.z_val]
731
732         distance = sqrt(
733             (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
734                 list1[2] - p_attuale[2]) ** 2)
735         time_required = (distance / list1[3])
736
737         time.sleep(time_required * 0.9)
738         a = True
739         while a:
740             if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < p_attuale[
741                 1] < (
742                     list1[1] + 2) and (
743                         list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
744                 a = False
745             else:
746                 time.sleep(0.1)

```

```

747
748         state = client.getMultirotorState(vehicle_name)
749         p_attuale = [state.kinematics_estimated.position.x_val,
750                     state.kinematics_estimated.position.y_val,
751                     state.kinematics_estimated.position.z_val]
752     elif (f_p_n < 0 and f_p_e < 0) or (f_p_n == 0 and f_p_e < 0):
753         if i % 2 == 0:
754             p = []
755             s = []
756             t = []
757             q = []
758             for num in range(1, i, 2):
759                 sott1 = num * ts
760                 a = f_p_n + sott1
761                 P = (a, f_p_e, h, v)
762                 p.append(P)
763                 S = (a, f_p_e + ll, h, v)
764                 s.append(S)
765             for mol in range(2, i + 1, 2):
766                 sott2 = mol * ts
767                 b = f_p_n + sott2
768                 T = (b, f_p_e + ll, h, v)
769                 t.append(T)
770                 Q = (b, f_p_e, h, v)
771                 q.append(Q)
772             Total = []
773             for n in range(len(q)):
774                 total = [p[n], s[n], t[n], q[n]]
775                 Total.append(total)
776             Totale = []
777             for n in range(len(Total)):
778                 for a in range(4):
779                     totale = Total[n][a]
780                     Totale.append(totale)
781             print(Totale)
782             for n in range(len(Totale)):
783                 client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3],
784                                           vehicle_name=vehicle_name)
785
786             list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
787             state = client.getMultirotorState(vehicle_name)
788             p_attuale = [state.kinematics_estimated.position.x_val,
789                         state.kinematics_estimated.position.y_val,
790                         state.kinematics_estimated.position.z_val]

```

```

791
792
793     distance = sqrt(
794         (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
795             list1[2] - p_attuale[2]) ** 2)
796     time_required = (distance / list1[3])
797
798     time.sleep(time_required * 0.9)
799     a = True
800     while a:
801         if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < p_attuale[
802             1] < (
803                 list1[1] + 2) and (
804                     list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
805             a = False
806         else:
807             time.sleep(0.1)
808
809         state = client.getMultirotorState(vehicle_name)
810         p_attuale = [state.kinematics_estimated.position.x_val,
811             state.kinematics_estimated.position.y_val,
812             state.kinematics_estimated.position.z_val]
813     else:
814         p = []
815         s = []
816         t = []
817         q = []
818         for num in range(1, i + 1, 2):
819             sott1 = num * ts
820             a = f_p_n + sott1
821             P = (a, f_p_e, h, v)
822             p.append(P)
823             S = (a, f_p_e + ll, h, v)
824             s.append(S)
825         for mol in range(2, i + 2, 2):
826             sott2 = mol * ts
827             b = f_p_n + sott2
828             T = (b, f_p_e + ll, h, v)
829             t.append(T)
830             Q = (b, f_p_e, h, v)
831             q.append(Q)
832         Total = []
833         for n in range(len(s)):
834             total = [p[n], s[n], t[n], q[n]]
835             Total.append(total)

```



```

835     Totale = []
836     for n in range(len(Total)):
837         for a in range(4):
838             totale = Total[n][a]
839             Totale.append(totale)
840     print(Totale)
841     limit = len(Totale) - 2
842     for n in range(limit):
843         client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3],
844                                   vehicle_name=vehicle_name)
845
846         list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
847         state = client.getMultirotorState(vehicle_name)
848         p_attuale = [state.kinematics_estimated.position.x_val,
849                     state.kinematics_estimated.position.y_val,
850                     state.kinematics_estimated.position.z_val]
851
852         distance = sqrt(
853             (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
854                 list1[2] - p_attuale[2]) ** 2)
855         time_required = (distance / list1[3])
856
857         time.sleep(time_required * 0.9)
858         a = True
859         while a:
860             if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < p_attuale[
861                 1] < (
862                     list1[1] + 2) and (
863                         list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
864                 a = False
865             else:
866                 time.sleep(0.1)
867
868             state = client.getMultirotorState(vehicle_name)
869             p_attuale = [state.kinematics_estimated.position.x_val,
870                         state.kinematics_estimated.position.y_val,
871                         state.kinematics_estimated.position.z_val]
872         if waypoints_data[first_child[n]]["leg_type"] == "fp_IterativeLeg":
873             name_next_point = waypoints_data[first_child[n]]["name_next_point"]
874             num_of_iter = waypoints_data[first_child[n]]["num_of_iter"]
875             name_point = first_child[n]
876             lip = waypoints_data[name_point]["len_iter_path"]
877             iter_path = []
878             for n in range(lip):

```

```

879         iter_path.append(waypoints_data[name_point]["name_point_iter_path" + str(n)])
880     rip = 0
881     while rip < num_of_iter:
882         for n in range(len(iter_path)):
883             name_point = iter_path[n]
884             if waypoints_data[name_point]["leg_type"] == "fp_TFLeg":
885                 n_coord = waypoints_data[name_point]["n_coord"]
886                 e_coord = waypoints_data[name_point]["e_coord"]
887                 alt = waypoints_data[name_point]["altitude"]
888                 speed = waypoints_data[name_point]["speed"]
889
890                 client.moveToPositionAsync(n_coord, e_coord, alt, speed, vehicle_name=vehicle_name)
891
892                 list1 = [n_coord, e_coord, alt, speed]
893                 state = client.getMultirotorState(vehicle_name)
894                 p_attuale = [state.kinematics_estimated.position.x_val,
895                             state.kinematics_estimated.position.y_val,
896                             state.kinematics_estimated.position.z_val]
897
898                 distance = sqrt(
899                     (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
900                         list1[2] - p_attuale[2]) ** 2)
901                 time_required = (distance / list1[3])
902
903                 time.sleep(time_required * 0.9)
904                 a = True
905                 while a:
906                     if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < p_attuale[
907                         1] < (
908                             list1[1] + 2) and (
909                                 list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
910                         a = False
911                     else:
912                         time.sleep(0.1)
913
914                     state = client.getMultirotorState(vehicle_name)
915                     p_attuale = [state.kinematics_estimated.position.x_val,
916                                 state.kinematics_estimated.position.y_val,
917                                 state.kinematics_estimated.position.z_val]
918                 if waypoints_data[name_point]["leg_type"] == "fp_Scan":
919                     ts = waypoints_data[name_point]["trackseparation"]
920                     f_p_n = waypoints_data[name_point]["point1_n"]
921                     f_p_e = waypoints_data[name_point]["point1_e"]
922                     v = waypoints_data[name_point]["speed"]

```

```

923 h = waypoints_data[name_point]["altitude"]
924 i = waypoints_data[name_point]["iteration"]
925 ll = waypoints_data[name_point]["length_side"]
926 client.moveToPositionAsync(f_p_n, f_p_e, h, v, vehicle_name=vehicle_name)
927
928 list1 = [f_p_n, f_p_e, h, v]
929 state = client.getMultirotorState(vehicle_name)
930 p_attuale = [state.kinematics_estimated.position.x_val,
931              state.kinematics_estimated.position.y_val,
932              state.kinematics_estimated.position.z_val]
933
934 distance = sqrt(
935     (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
936         list1[2] - p_attuale[2]) ** 2)
937 time_required = (distance / list1[3])
938
939 time.sleep(time_required * 0.9)
940 a = True
941 while a:
942     if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < p_attuale[
943         1] < (
944             list1[1] + 2) and (
945                 list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
946         a = False
947     else:
948         time.sleep(0.1)
949
950     state = client.getMultirotorState(vehicle_name)
951     p_attuale = [state.kinematics_estimated.position.x_val,
952                  state.kinematics_estimated.position.y_val,
953                  state.kinematics_estimated.position.z_val]
954 if (f_p_e < 0 < f_p_n) or (f_p_n > 0 and f_p_e == 0):
955     if i % 2 == 0:
956         p = []
957         s = []
958         t = []
959         q = []
960         for num in range(1, i, 2):
961             sott1 = num * ts
962             a = f_p_n - sott1
963             p = (a, f_p_e, h, v)
964             p.append(P)
965             S = (a, f_p_e + ll, h, v)
966             s.append(S)

```

```

967     for mol in range(2, i + 1, 2):
968         sott2 = mol * ts
969         b = f_p_n - sott2
970         T = (b, f_p_e + ll, h, v)
971         t.append(T)
972         Q = (b, f_p_e, h, v)
973         q.append(Q)
974     Total = []
975     for n in range(len(q)):
976         total = [p[n], s[n], t[n], q[n]]
977         Total.append(total)
978     Totale = []
979     for n in range(len(Total)):
980         for a in range(4):
981             totale = Total[n][a]
982             Totale.append(totale)
983     print(Totale)
984     for n in range(len(Totale)):
985         client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2],
986                                   Totale[n][3],
987                                   vehicle_name=vehicle_name)
988
989     list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
990     state = client.getMultirotorState(vehicle_name)
991     p_attuale = [state.kinematics_estimated.position.x_val,
992                 state.kinematics_estimated.position.y_val,
993                 state.kinematics_estimated.position.z_val]
994
995     distance = sqrt(
996         (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
997             list1[2] - p_attuale[2]) ** 2)
998     time_required = (distance / list1[3])
999
1000    time.sleep(time_required * 0.9)
1001    a = True
1002    while a:
1003        if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < \
1004            p_attuale[
1005                1] < (
1006                    list1[1] + 2) and (
1007                        list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
1008            a = False
1009        else:
1010            time.sleep(0.1)

```

```

1011
1012
1013         state = client.getMultirotorState(vehicle_name)
1014         p_attuale = [state.kinematics_estimated.position.x_val,
1015                     state.kinematics_estimated.position.y_val,
1016                     state.kinematics_estimated.position.z_val]
1017
1018     else:
1019         p = []
1020         s = []
1021         t = []
1022         q = []
1023         for num in range(1, i + 1, 2):
1024             sott1 = num * ts
1025             a = f_p_n - sott1
1026             P = (a, f_p_e, h, v)
1027             p.append(P)
1028             S = (a, f_p_e + ll, h, v)
1029             s.append(S)
1030         for mol in range(2, i + 2, 2):
1031             sott2 = mol * ts
1032             b = f_p_n - sott2
1033             T = (b, f_p_e + ll, h, v)
1034             t.append(T)
1035             Q = (b, f_p_e, h, v)
1036             q.append(Q)
1037         Total = []
1038         for n in range(len(s)):
1039             total = [p[n], s[n], t[n], q[n]]
1040             Total.append(total)
1041         Totale = []
1042         for n in range(len(Total)):
1043             for a in range(4):
1044                 totale = Total[n][a]
1045                 Totale.append(totale)
1046         print(Totale)
1047         limit = len(Totale) - 2
1048         for n in range(limit):
1049             client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2],
1050                                     Totale[n][3],
1051                                     vehicle_name=vehicle_name)
1052
1053         list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
1054         state = client.getMultirotorState(vehicle_name)
1055         p_attuale = [state.kinematics_estimated.position.x_val,
1056                     state.kinematics_estimated.position.y_val,

```

```

1055         state.kinematics_estimated.position.z_val]
1056
1057     distance = sqrt(
1058         (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
1059             list1[2] - p_attuale[2]) ** 2)
1060     time_required = (distance / list1[3])
1061
1062     time.sleep(time_required * 0.9)
1063     a = True
1064     while a:
1065         if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < \
1066             p_attuale[
1067                 1] < (
1068                     list1[1] + 2) and (
1069                         list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
1070             a = False
1071         else:
1072             time.sleep(0.1)
1073
1074         state = client.getMultirotorState(vehicle_name)
1075         p_attuale = [state.kinematics_estimated.position.x_val,
1076                     state.kinematics_estimated.position.y_val,
1077                     state.kinematics_estimated.position.z_val]
1078     elif (f_p_n > 0 and f_p_e > 0) or (f_p_n == 0 and f_p_e > 0):
1079         if i % 2 == 0:
1080             p = []
1081             s = []
1082             t = []
1083             q = []
1084             for num in range(1, i, 2):
1085                 sott1 = num * ts
1086                 a = f_p_n - sott1
1087                 p = (a, f_p_e, h, v)
1088                 p.append(p)
1089                 s = (a, f_p_e - ll, h, v)
1090                 s.append(s)
1091             for mol in range(2, i + 1, 2):
1092                 sott2 = mol * ts
1093                 b = f_p_n - sott2
1094                 t = (b, f_p_e - ll, h, v)
1095                 t.append(t)
1096                 q = (b, f_p_e, h, v)
1097                 q.append(q)
1098         Total = []

```

```

11099 for n in range(len(q)):
11100     total = [p[n], s[n], t[n], q[n]]
11101     Total.append(total)
11102 Total = []
11103 for n in range(len(Total)):
11104     for a in range(4):
11105         totale = Total[n][a]
11106         Totale.append(totale)
11107 print(Totale)
11108 for n in range(len(Totale)):
11109     client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2],
11110                               Totale[n][3],
11111                               vehicle_name=vehicle_name)
11112
11113     list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
11114     state = client.getMultirotorState(vehicle_name)
11115     p_attuale = [state.kinematics_estimated.position.x_val,
11116                  state.kinematics_estimated.position.y_val,
11117                  state.kinematics_estimated.position.z_val]
11118
11119     distance = sqrt(
11120         (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
11121             list1[2] - p_attuale[2]) ** 2)
11122     time_required = (distance / list1[3])
11123
11124     time.sleep(time_required * 0.9)
11125     a = True
11126     while a:
11127         if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < \
11128             p_attuale[
11129                 1] < (
11130                     list1[1] + 2) and (
11131                         list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
11132             a = False
11133         else:
11134             time.sleep(0.1)
11135
11136         state = client.getMultirotorState(vehicle_name)
11137         p_attuale = [state.kinematics_estimated.position.x_val,
11138                      state.kinematics_estimated.position.y_val,
11139                      state.kinematics_estimated.position.z_val]
11140     else:
11141         p = []
11142         s = []

```

```

1143 t = []
1144 q = []
1145 for num in range(1, i + 1, 2):
1146     sott1 = num * ts
1147     a = f_p_n - sott1
1148     p = (a, f_p_e, h, v)
1149     p.append(P)
1150     s = (a, f_p_e - ll, h, v)
1151     s.append(S)
1152 for mol in range(2, i + 2, 2):
1153     sott2 = mol * ts
1154     b = f_p_n - sott2
1155     T = (b, f_p_e - ll, h, v)
1156     t.append(T)
1157     Q = (b, f_p_e, h, v)
1158     q.append(Q)
1159 Total = []
1160 for n in range(len(s)):
1161     total = [p[n], s[n], t[n], q[n]]
1162     Total.append(total)
1163 Totale = []
1164 for n in range(len(Total)):
1165     for a in range(4):
1166         totale = Total[n][a]
1167         Totale.append(totale)
1168 print(Totale)
1169 limit = len(Totale) - 2
1170 for n in range(limit):
1171     client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2],
1172                               Totale[n][3],
1173                               vehicle_name=vehicle_name)
1174
1175 list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
1176 state = client.getMultirotorState(vehicle_name)
1177 p_attuale = [state.kinematics_estimated.position.x_val,
1178              state.kinematics_estimated.position.y_val,
1179              state.kinematics_estimated.position.z_val]
1180
1181 distance = sqrt(
1182     (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
1183         list1[2] - p_attuale[2]) ** 2)
1184 time_required = (distance / list1[3])
1185
1186 time.sleep(time_required * 0.9)

```



```

1187         a = True
1188         while a:
1189             if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < \
1190                 p_attuale[
1191                     1] < (
1192                         list1[1] + 2) and (
1193                             list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
1194                 a = False
1195             else:
1196                 time.sleep(0.1)
1197
1198                 state = client.getMultirotorState(vehicle_name)
1199                 p_attuale = [state.kinematics_estimated.position.x_val,
1200                             state.kinematics_estimated.position.y_val,
1201                             state.kinematics_estimated.position.z_val]
1202
1203         elif (f_p_n < 0 < f_p_e) or (f_p_n < 0 and f_p_e == 0):
1204             if i % 2 == 0:
1205                 p = []
1206                 s = []
1207                 t = []
1208                 q = []
1209                 for num in range(1, i, 2):
1210                     sott1 = num * ts
1211                     a = f_p_n + sott1
1212                     p = (a, f_p_e, h, v)
1213                     p.append(p)
1214                     s = (a, f_p_e - ll, h, v)
1215                     s.append(s)
1216                 for mol in range(2, i + 1, 2):
1217                     sott2 = mol * ts
1218                     b = f_p_n + sott2
1219                     t = (b, f_p_e - ll, h, v)
1220                     t.append(t)
1221                     q = (b, f_p_e, h, v)
1222                     q.append(q)
1223                 Total = []
1224                 for n in range(len(q)):
1225                     total = [p[n], s[n], t[n], q[n]]
1226                     Total.append(total)
1227                 Totale = []
1228                 for n in range(len(Total)):
1229                     for a in range(4):
1230                         totale = Total[n][a]
1231                         Totale.append(totale)

```

```

1231 print(Totale)
1232 for n in range(len(Totale)):
1233     client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2],
1234                               Totale[n][3],
1235                               vehicle_name=vehicle_name)
1236
1237     list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
1238     state = client.getMultirotorState(vehicle_name)
1239     p_attuale = [state.kinematics_estimated.position.x_val,
1240                 state.kinematics_estimated.position.y_val,
1241                 state.kinematics_estimated.position.z_val]
1242
1243     distance = sqrt(
1244         (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
1245             list1[2] - p_attuale[2]) ** 2)
1246     time_required = (distance / list1[3])
1247
1248     time.sleep(time_required * 0.9)
1249     a = True
1250     while a:
1251         if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < \
1252             p_attuale[1] < (
1253                 list1[1] + 2) and (
1254                     list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
1255             a = False
1256         else:
1257             time.sleep(0.1)
1258
1259         state = client.getMultirotorState(vehicle_name)
1260         p_attuale = [state.kinematics_estimated.position.x_val,
1261                     state.kinematics_estimated.position.y_val,
1262                     state.kinematics_estimated.position.z_val]
1263
1264     else:
1265         p = []
1266         s = []
1267         t = []
1268         q = []
1269         for num in range(1, i + 1, 2):
1270             sott1 = num * ts
1271             a = f_p_n + sott1
1272             p = (a, f_p_e, h, v)
1273             p.append(P)
1274             S = (a, f_p_e - ll, h, v)

```

```

1275         s.append(S)
1276     for mol in range(2, i + 2, 2):
1277         sott2 = mol * ts
1278         b = f_p_n + sott2
1279         T = (b, f_p_e - ll, h, v)
1280         t.append(T)
1281         Q = (b, f_p_e, h, v)
1282         q.append(Q)
1283     Total = []
1284     for n in range(len(s)):
1285         total = [p[n], s[n], t[n], q[n]]
1286         Total.append(total)
1287     Totale = []
1288     for n in range(len(Total)):
1289         for a in range(4):
1290             totale = Total[n][a]
1291             Totale.append(totale)
1292     limit = len(Totale) - 2
1293     for n in range(limit):
1294         client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2],
1295                                   Totale[n][3],
1296                                   vehicle_name=vehicle_name)
1297
1298     list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
1299     state = client.getMultirotorState(vehicle_name)
1300     p_attuale = [state.kinematics_estimated.position.x_val,
1301                 state.kinematics_estimated.position.y_val,
1302                 state.kinematics_estimated.position.z_val]
1303
1304     distance = sqrt(
1305         (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
1306             list1[2] - p_attuale[2]) ** 2)
1307     time_required = (distance / list1[3])
1308
1309     time.sleep(time_required * 0.9)
1310     a = True
1311     while a:
1312         if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < \
1313             p_attuale[
1314                 1] < (
1315                     list1[1] + 2) and (
1316                         list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
1317             a = False
1318     else:

```

```

1319         time.sleep(0.1)
1320
1321         state = client.getMultirotorState(vehicle_name)
1322         p_attuale = [state.kinematics_estimated.position.x_val,
1323                     state.kinematics_estimated.position.y_val,
1324                     state.kinematics_estimated.position.z_val]
1325     elif (f_p_n < 0 and f_p_e < 0) or (f_p_n == 0 and f_p_e < 0):
1326         if i % 2 == 0:
1327             p = []
1328             s = []
1329             t = []
1330             q = []
1331             for num in range(1, i, 2):
1332                 sott1 = num * ts
1333                 a = f_p_n + sott1
1334                 p = (a, f_p_e, h, v)
1335                 p.append(p)
1336                 s = (a, f_p_e + ll, h, v)
1337                 s.append(s)
1338             for mol in range(2, i + 1, 2):
1339                 sott2 = mol * ts
1340                 b = f_p_n + sott2
1341                 t = (b, f_p_e + ll, h, v)
1342                 t.append(t)
1343                 q = (b, f_p_e, h, v)
1344                 q.append(q)
1345             Total = []
1346             for n in range(len(q)):
1347                 total = [p[n], s[n], t[n], q[n]]
1348                 Total.append(total)
1349             Totale = []
1350             for n in range(len(Total)):
1351                 for a in range(4):
1352                     totale = Total[n][a]
1353                     Totale.append(totale)
1354             print(Totale)
1355             for n in range(len(Totale)):
1356                 client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2],
1357                                           Totale[n][3],
1358                                           vehicle_name=vehicle_name)
1359
1360             list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
1361             state = client.getMultirotorState(vehicle_name)
1362             p_attuale = [state.kinematics_estimated.position.x_val,

```

```

1363         state.kinematics_estimated.position.y_val,
1364         state.kinematics_estimated.position.z_val]
1365
1366     distance = sqrt(
1367         (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
1368             list1[2] - p_attuale[2]) ** 2)
1369     time_required = (distance / list1[3])
1370
1371     time.sleep(time_required * 0.9)
1372     a = True
1373     while a:
1374         if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < \
1375             p_attuale[
1376                 1] < (
1377                     list1[1] + 2) and (
1378                         list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
1379             a = False
1380         else:
1381             time.sleep(0.1)
1382
1383         state = client.getMultirotorState(vehicle_name)
1384         p_attuale = [state.kinematics_estimated.position.x_val,
1385             state.kinematics_estimated.position.y_val,
1386             state.kinematics_estimated.position.z_val]
1387     else:
1388         p = []
1389         s = []
1390         t = []
1391         q = []
1392         for num in range(1, i + 1, 2):
1393             sott1 = num * ts
1394             a = f_p_n + sott1
1395             p = (a, f_p_e, h, v)
1396             p.append(P)
1397             s = (a, f_p_e + ll, h, v)
1398             s.append(S)
1399         for mol in range(2, i + 2, 2):
1400             sott2 = mol * ts
1401             b = f_p_n + sott2
1402             T = (b, f_p_e + ll, h, v)
1403             t.append(T)
1404             Q = (b, f_p_e, h, v)
1405             q.append(Q)
1406         Total = []

```

```

1407     for n in range(len(s)):
1408         total = [p[n], s[n], t[n], q[n]]
1409         Total.append(total)
1410     Totale = []
1411     for n in range(len(Total)):
1412         for a in range(4):
1413             totale = Total[n][a]
1414             Totale.append(totale)
1415     print(Totale)
1416     limit = len(Totale) - 2
1417     for n in range(limit):
1418         client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2],
1419                                   Totale[n][3],
1420                                   vehicle_name=vehicle_name)
1421
1422         list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
1423         state = client.getMultirotorState(vehicle_name)
1424         p_attuale = [state.kinematics_estimated.position.x_val,
1425                     state.kinematics_estimated.position.y_val,
1426                     state.kinematics_estimated.position.z_val]
1427
1428         distance = sqrt(
1429             (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
1430                 list1[2] - p_attuale[2]) ** 2)
1431         time_required = (distance / list1[3])
1432
1433         time.sleep(time_required * 0.9)
1434         a = True
1435         while a:
1436             if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < \
1437                 p_attuale[
1438                     1] < (
1439                     list1[1] + 2) and (
1440                         list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
1441                 a = False
1442             else:
1443                 time.sleep(0.1)
1444
1445             state = client.getMultirotorState(vehicle_name)
1446             p_attuale = [state.kinematics_estimated.position.x_val,
1447                         state.kinematics_estimated.position.y_val,
1448                         state.kinematics_estimated.position.z_val]
1449
1450         rip += 1
1450         n_coord = waypoints_data[name_next_point]["n_coord"]

```

```

1451 e_coord = waypoints_data[name_next_point]["e_coord"]
1452 h = waypoints_data[name_next_point]["altitude"]
1453 v = waypoints_data[name_next_point]["speed"]
1454
1455 client.moveToPositionAsync(n_coord, e_coord, h, v, vehicle_name=vehicle_name)
1456
1457 list1 = [n_coord, e_coord, h, v]
1458 state = client.getMultirotorState(vehicle_name)
1459 p_attuale = [state.kinematics_estimated.position.x_val, state.kinematics_estimated.position.y_val,
1460              state.kinematics_estimated.position.z_val]
1461
1462 distance = sqrt(
1463     (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
1464         list1[2] - p_attuale[2]) ** 2)
1465 time_required = (distance / list1[3])
1466
1467 time.sleep(time_required * 0.9)
1468 a = True
1469 while a:
1470     if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < p_attuale[1] < (
1471         list1[1] + 2) and (
1472             list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
1473         a = False
1474     else:
1475         time.sleep(0.1)
1476
1477     state = client.getMultirotorState(vehicle_name)
1478     p_attuale = [state.kinematics_estimated.position.x_val,
1479                 state.kinematics_estimated.position.y_val,
1480                 state.kinematics_estimated.position.z_val]
1481 if waypoints_data[first_child[n]]["leg_type"] == "fp_IntersectionLeg":
1482     limit = waypoints_data[first_child[n]]["limit"]
1483     option = input("Intersection Leg: to select one of the possibilities, insert a "
1484                   "number included between 1 and " + str(limit) + ": ")
1485     option = int(option)
1486     name_points = first_child[n]
1487     name_point = waypoints_data[name_points]["name_point_possibility" + str(option - 1)]
1488     if waypoints_data[name_point]["leg_type"] == "fp_TFLeg":
1489         n_coord = waypoints_data[name_point]["n_coord"]
1490         e_coord = waypoints_data[name_point]["e_coord"]
1491         alt = waypoints_data[name_point]["altitude"]
1492         speed = waypoints_data[name_point]["speed"]
1493
1494     client.moveToPositionAsync(n_coord, e_coord, alt, speed, vehicle_name=vehicle_name)

```

```

1495
1496     list1 = [n_coord, e_coord, alt, speed]
1497     state = client.getMultirotorState(vehicle_name)
1498     p_attuale = [state.kinematics_estimated.position.x_val,
1499                 state.kinematics_estimated.position.y_val,
1500                 state.kinematics_estimated.position.z_val]
1501
1502     distance = sqrt(
1503         (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
1504             list1[2] - p_attuale[2]) ** 2)
1505     time_required = (distance / list1[3])
1506
1507     time.sleep(time_required * 0.9)
1508     a = True
1509     while a:
1510         if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < p_attuale[1] < (
1511             list1[1] + 2) and (
1512                 list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
1513             a = False
1514         else:
1515             time.sleep(0.1)
1516
1517         state = client.getMultirotorState(vehicle_name)
1518         p_attuale = [state.kinematics_estimated.position.x_val,
1519                     state.kinematics_estimated.position.y_val,
1520                     state.kinematics_estimated.position.z_val]
1521     if waypoints_data[name_point]["leg_type"] == "fp_Scan":
1522         ts = waypoints_data[name_point]["trackseparation"]
1523         f_p_n = waypoints_data[name_point]["point1_n"]
1524         f_p_e = waypoints_data[name_point]["point1_e"]
1525         v = waypoints_data[name_point]["speed"]
1526         h = waypoints_data[name_point]["altitude"]
1527         i = waypoints_data[name_point]["iteration"]
1528         ll = waypoints_data[name_point]["length_side"]
1529         client.moveToPositionAsync(f_p_n, f_p_e, h, v, vehicle_name=vehicle_name)
1530
1531         list1 = [f_p_n, f_p_e, h, v]
1532         state = client.getMultirotorState(vehicle_name)
1533         p_attuale = [state.kinematics_estimated.position.x_val,
1534                     state.kinematics_estimated.position.y_val,
1535                     state.kinematics_estimated.position.z_val]
1536
1537         distance = sqrt(
1538             (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (

```



```

1539         list1[2] - p_attuale[2]) ** 2)
1540     time_required = (distance / list1[3])
1541
1542     time.sleep(time_required * 0.9)
1543     a = True
1544     while a:
1545         if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < p_attuale[1] < (
1546             list1[1] + 2) and (
1547                 list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
1548             a = False
1549         else:
1550             time.sleep(0.1)
1551
1552         state = client.getMultirotorState(vehicle_name)
1553         p_attuale = [state.kinematics_estimated.position.x_val,
1554                     state.kinematics_estimated.position.y_val,
1555                     state.kinematics_estimated.position.z_val]
1556     if (f_p_e < 0 < f_p_n) or (f_p_n > 0 and f_p_e == 0):
1557         if i % 2 == 0:
1558             p = []
1559             s = []
1560             t = []
1561             q = []
1562             for num in range(1, i, 2):
1563                 sott1 = num * ts
1564                 a = f_p_n - sott1
1565                 P = (a, f_p_e, h, v)
1566                 p.append(P)
1567                 S = (a, f_p_e + ll, h, v)
1568                 s.append(S)
1569             for mol in range(2, i + 1, 2):
1570                 sott2 = mol * ts
1571                 b = f_p_n - sott2
1572                 T = (b, f_p_e + ll, h, v)
1573                 t.append(T)
1574                 Q = (b, f_p_e, h, v)
1575                 q.append(Q)
1576             Total = []
1577             for n in range(len(q)):
1578                 total = [p[n], s[n], t[n], q[n]]
1579                 Total.append(total)
1580             Totale = []
1581             for n in range(len(Total)):
1582                 for a in range(4):

```

```

1583         totale = Total[n][a]
1584         Totale.append(totale)
1585     print(Totale)
1586     for n in range(len(Totale)):
1587         client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3],
1588                                   vehicle_name=vehicle_name)
1589
1590         list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
1591         state = client.getMultirotorState(vehicle_name)
1592         p_attuale = [state.kinematics_estimated.position.x_val,
1593                     state.kinematics_estimated.position.y_val,
1594                     state.kinematics_estimated.position.z_val]
1595
1596         distance = sqrt(
1597             (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
1598                 list1[2] - p_attuale[2]) ** 2)
1599         time_required = (distance / list1[3])
1600
1601         time.sleep(time_required * 0.9)
1602         a = True
1603         while a:
1604             if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < \
1605                 p_attuale[1] < (
1606                     list1[1] + 2) and (
1607                         list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
1608                 a = False
1609             else:
1610                 time.sleep(0.1)
1611
1612             state = client.getMultirotorState(vehicle_name)
1613             p_attuale = [state.kinematics_estimated.position.x_val,
1614                         state.kinematics_estimated.position.y_val,
1615                         state.kinematics_estimated.position.z_val]
1616
1617     else:
1618         p = []
1619         s = []
1620         t = []
1621         q = []
1622         for num in range(1, i + 1, 2):
1623             sott1 = num * ts
1624             a = f_p_n - sott1
1625             P = (a, f_p_e, h, v)
1626             p.append(P)

```

```

1627     S = (a, f_p_e + ll, h, v)
1628     s.append(S)
1629     for mol in range(2, i + 2, 2):
1630         sott2 = mol * ts
1631         b = f_p_n - sott2
1632         T = (b, f_p_e + ll, h, v)
1633         t.append(T)
1634         Q = (b, f_p_e, h, v)
1635         q.append(Q)
1636     Total = []
1637     for n in range(len(s)):
1638         total = [p[n], s[n], t[n], q[n]]
1639         Total.append(total)
1640     Totale = []
1641     for n in range(len(Total)):
1642         for a in range(4):
1643             totale = Total[n][a]
1644             Totale.append(totale)
1645     print(Totale)
1646     limit = len(Totale) - 2
1647     for n in range(limit):
1648         client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3],
1649                                   vehicle_name=vehicle_name)
1650
1651         list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
1652         state = client.getMultirotorState(vehicle_name)
1653         p_attuale = [state.kinematics_estimated.position.x_val,
1654                     state.kinematics_estimated.position.y_val,
1655                     state.kinematics_estimated.position.z_val]
1656
1657         distance = sqrt(
1658             (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
1659                 list1[2] - p_attuale[2]) ** 2)
1660         time_required = (distance / list1[3])
1661
1662         time.sleep(time_required * 0.9)
1663         a = True
1664         while a:
1665             if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < \
1666                 p_attuale[
1667                     1] < (
1668                         list1[1] + 2) and (
1669                             list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
1670                 a = False

```

```

1671         else:
1672             time.sleep(0.1)
1673
1674             state = client.getMultirotorState(vehicle_name)
1675             p_attuale = [state.kinematics_estimated.position.x_val,
1676                         state.kinematics_estimated.position.y_val,
1677                         state.kinematics_estimated.position.z_val]
1678
1679             elif (f_p_n > 0 and f_p_e > 0) or (f_p_n == 0 and f_p_e > 0):
1680                 if i % 2 == 0:
1681                     p = []
1682                     s = []
1683                     t = []
1684                     q = []
1685                     for num in range(1, i, 2):
1686                         sott1 = num * ts
1687                         a = f_p_n - sott1
1688                         P = (a, f_p_e, h, v)
1689                         p.append(P)
1690                         S = (a, f_p_e - ll, h, v)
1691                         s.append(S)
1692                     for mol in range(2, i + 1, 2):
1693                         sott2 = mol * ts
1694                         b = f_p_n - sott2
1695                         T = (b, f_p_e - ll, h, v)
1696                         t.append(T)
1697                         Q = (b, f_p_e, h, v)
1698                         q.append(Q)
1699                     Total = []
1700                     for n in range(len(q)):
1701                         total = [p[n], s[n], t[n], q[n]]
1702                         Total.append(total)
1703                     Totale = []
1704                     for n in range(len(Total)):
1705                         for a in range(4):
1706                             totale = Total[n][a]
1707                             Totale.append(totale)
1708                     print(Totale)
1709                     for n in range(len(Totale)):
1710                         client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3],
1711                                                     vehicle_name=vehicle_name)
1712
1713                     list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
1714                     state = client.getMultirotorState(vehicle_name)
1715                     p_attuale = [state.kinematics_estimated.position.x_val,

```

```

1715         state.kinematics_estimated.position.y_val,
1716         state.kinematics_estimated.position.z_val]
1717
1718     distance = sqrt(
1719         (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
1720             list1[2] - p_attuale[2]) ** 2)
1721     time_required = (distance / list1[3])
1722
1723     time.sleep(time_required * 0.9)
1724     a = True
1725     while a:
1726         if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < \
1727             p_attuale[
1728                 1] < (
1729                     list1[1] + 2) and (
1730                         list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
1731             a = False
1732         else:
1733             time.sleep(0.1)
1734
1735         state = client.getMultirotorState(vehicle_name)
1736         p_attuale = [state.kinematics_estimated.position.x_val,
1737                     state.kinematics_estimated.position.y_val,
1738                     state.kinematics_estimated.position.z_val]
1739     else:
1740         p = []
1741         s = []
1742         t = []
1743         q = []
1744         for num in range(1, i + 1, 2):
1745             sott1 = num * ts
1746             a = f_p_n - sott1
1747             p = (a, f_p_e, h, v)
1748             p.append(P)
1749             s = (a, f_p_e - ll, h, v)
1750             s.append(S)
1751         for mol in range(2, i + 2, 2):
1752             sott2 = mol * ts
1753             b = f_p_n - sott2
1754             T = (b, f_p_e - ll, h, v)
1755             t.append(T)
1756             Q = (b, f_p_e, h, v)
1757             q.append(Q)
1758         Total = []

```

```

1759         for n in range(len(s)):
1760             total = [p[n], s[n], t[n], q[n]]
1761             Total.append(total)
1762         Totale = []
1763         for n in range(len(Total)):
1764             for a in range(4):
1765                 totale = Total[n][a]
1766                 Totale.append(totale)
1767         print(Totale)
1768         limit = len(Totale) - 2
1769         for n in range(limit):
1770             client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3],
1771                                       vehicle_name=vehicle_name)
1772
1773             list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
1774             state = client.getMultirotorState(vehicle_name)
1775             p_attuale = [state.kinematics_estimated.position.x_val,
1776                         state.kinematics_estimated.position.y_val,
1777                         state.kinematics_estimated.position.z_val]
1778
1779             distance = sqrt(
1780                 (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
1781                     list1[2] - p_attuale[2]) ** 2)
1782             time_required = (distance / list1[3])
1783
1784             time.sleep(time_required * 0.9)
1785             a = True
1786             while a:
1787                 if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < \
1788                     p_attuale[
1789                         1] < (
1790                             list1[1] + 2) and (
1791                                 list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
1792                     a = False
1793             else:
1794                 time.sleep(0.1)
1795
1796                 state = client.getMultirotorState(vehicle_name)
1797                 p_attuale = [state.kinematics_estimated.position.x_val,
1798                             state.kinematics_estimated.position.y_val,
1799                             state.kinematics_estimated.position.z_val]
1800             elif (f_p_n < 0 < f_p_e) or (f_p_n < 0 and f_p_e == 0):
1801                 if i % 2 == 0:
1802                     p = []

```

```

1803 s = []
1804 t = []
1805 q = []
1806 for num in range(1, i, 2):
1807     sott1 = num * ts
1808     a = f_p_n + sott1
1809     p = (a, f_p_e, h, v)
1810     p.append(P)
1811     S = (a, f_p_e - ll, h, v)
1812     s.append(S)
1813 for mol in range(2, i + 1, 2):
1814     sott2 = mol * ts
1815     b = f_p_n + sott2
1816     T = (b, f_p_e - ll, h, v)
1817     t.append(T)
1818     Q = (b, f_p_e, h, v)
1819     q.append(Q)
1820 Total = []
1821 for n in range(len(q)):
1822     total = [p[n], s[n], t[n], q[n]]
1823     Total.append(total)
1824 Totale = []
1825 for n in range(len(Total)):
1826     for a in range(4):
1827         totale = Total[n][a]
1828         Totale.append(totale)
1829 print(Totale)
1830 for n in range(len(Totale)):
1831     client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3],
1832                               vehicle_name=vehicle_name)
1833
1834     list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
1835     state = client.getMultirotorState(vehicle_name)
1836     p_attuale = [state.kinematics_estimated.position.x_val,
1837                  state.kinematics_estimated.position.y_val,
1838                  state.kinematics_estimated.position.z_val]
1839
1840     distance = sqrt(
1841         (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
1842             list1[2] - p_attuale[2]) ** 2)
1843     time_required = (distance / list1[3])
1844
1845     time.sleep(time_required * 0.9)
1846     a = True

```

```

1847 while a:
1848     if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < \
1849         p_attuale[1] < (
1850             list1[1] + 2) and (
1851                 list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
1852         a = False
1853     else:
1854         time.sleep(0.1)
1855
1856     state = client.getMultirotorState(vehicle_name)
1857     p_attuale = [state.kinematics_estimated.position.x_val,
1858                 state.kinematics_estimated.position.y_val,
1859                 state.kinematics_estimated.position.z_val]
1860
1861 else:
1862     p = []
1863     s = []
1864     t = []
1865     q = []
1866     for num in range(1, i + 1, 2):
1867         sott1 = num * ts
1868         a = f_p_n + sott1
1869         P = (a, f_p_e, h, v)
1870         p.append(P)
1871         S = (a, f_p_e - ll, h, v)
1872         s.append(S)
1873     for mol in range(2, i + 2, 2):
1874         sott2 = mol * ts
1875         b = f_p_n + sott2
1876         T = (b, f_p_e - ll, h, v)
1877         t.append(T)
1878         Q = (b, f_p_e, h, v)
1879         q.append(Q)
1880     Total = []
1881     for n in range(len(s)):
1882         total = [p[n], s[n], t[n], q[n]]
1883         Total.append(total)
1884     Totale = []
1885     for n in range(len(Total)):
1886         for a in range(4):
1887             totale = Total[n][a]
1888             Totale.append(totale)
1889     limit = len(Totale) - 2
1890     for n in range(limit):

```



```

1891 client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3],
1892                             vehicle_name=vehicle_name)
1893
1894 list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
1895 state = client.getMultirotorState(vehicle_name)
1896 p_attuale = [state.kinematics_estimated.position.x_val,
1897               state.kinematics_estimated.position.y_val,
1898               state.kinematics_estimated.position.z_val]
1899
1900 distance = sqrt(
1901     (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
1902         list1[2] - p_attuale[2]) ** 2)
1903 time_required = (distance / list1[3])
1904
1905 time.sleep(time_required * 0.9)
1906 a = True
1907 while a:
1908     if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < \
1909         p_attuale[
1910             1] < (
1911                 list1[1] + 2) and (
1912                     list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
1913         a = False
1914     else:
1915         time.sleep(0.1)
1916
1917     state = client.getMultirotorState(vehicle_name)
1918     p_attuale = [state.kinematics_estimated.position.x_val,
1919                 state.kinematics_estimated.position.y_val,
1920                 state.kinematics_estimated.position.z_val]
1921 elif (f_p_n < 0 and f_p_e < 0) or (f_p_n == 0 and f_p_e < 0):
1922     if i % 2 == 0:
1923         p = []
1924         s = []
1925         t = []
1926         q = []
1927         for num in range(1, i, 2):
1928             sott1 = num * ts
1929             a = f_p_n + sott1
1930             P = (a, f_p_e, h, v)
1931             p.append(P)
1932             S = (a, f_p_e + 11, h, v)
1933             s.append(S)
1934         for mol in range(2, i + 1, 2):

```

```

1935         sott2 = mol * ts
1936         b = f_p_n + sott2
1937         T = (b, f_p_e + ll, h, v)
1938         t.append(T)
1939         Q = (b, f_p_e, h, v)
1940         q.append(Q)
1941         Total = []
1942         for n in range(len(q)):
1943             total = [p[n], s[n], t[n], q[n]]
1944             Total.append(total)
1945         Totale = []
1946         for n in range(len(Total)):
1947             for a in range(4):
1948                 totale = Total[n][a]
1949                 Totale.append(totale)
1950         print(Totale)
1951         for n in range(len(Totale)):
1952             client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3],
1953                                       vehicle_name=vehicle_name)
1954
1955         list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
1956         state = client.getMultirotorState(vehicle_name)
1957         p_attuale = [state.kinematics_estimated.position.x_val,
1958                     state.kinematics_estimated.position.y_val,
1959                     state.kinematics_estimated.position.z_val]
1960
1961         distance = sqrt(
1962             (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
1963                 list1[2] - p_attuale[2]) ** 2)
1964         time_required = (distance / list1[3])
1965
1966         time.sleep(time_required * 0.9)
1967         a = True
1968         while a:
1969             if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < \
1970                 p_attuale[
1971                     1] < (
1972                         list1[1] + 2) and (
1973                             list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
1974                 a = False
1975             else:
1976                 time.sleep(0.1)
1977
1978         state = client.getMultirotorState(vehicle_name)

```

```

1979         p_attuale = [state.kinematics_estimated.position.x_val,
1980                        state.kinematics_estimated.position.y_val,
1981                        state.kinematics_estimated.position.z_val]
1982
1983     else:
1984         p = []
1985         s = []
1986         t = []
1987         q = []
1988         for num in range(1, i + 1, 2):
1989             sott1 = num * ts
1990             a = f_p_n + sott1
1991             P = (a, f_p_e, h, v)
1992             p.append(P)
1993             S = (a, f_p_e + ll, h, v)
1994             s.append(S)
1995         for mol in range(2, i + 2, 2):
1996             sott2 = mol * ts
1997             b = f_p_n + sott2
1998             T = (b, f_p_e + ll, h, v)
1999             t.append(T)
2000             Q = (b, f_p_e, h, v)
2001             q.append(Q)
2002         Total = []
2003         for n in range(len(s)):
2004             total = [p[n], s[n], t[n], q[n]]
2005             Total.append(total)
2006         Totale = []
2007         for n in range(len(Total)):
2008             for a in range(4):
2009                 totale = Total[n][a]
2010                 Totale.append(totale)
2011         print(Totale)
2012         limit = len(Totale) - 2
2013         for n in range(limit):
2014             client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3],
2015                                       vehicle_name=vehicle_name)
2016
2017         list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
2018         state = client.getMultirotorState(vehicle_name)
2019         p_attuale = [state.kinematics_estimated.position.x_val,
2020                      state.kinematics_estimated.position.y_val,
2021                      state.kinematics_estimated.position.z_val]
2022
2023         distance = sqrt(

```

```

2023         (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
2024             list1[2] - p_attuale[2]) ** 2)
2025         time_required = (distance / list1[3])
2026
2027         time.sleep(time_required * 0.9)
2028         a = True
2029         while a:
2030             if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < \
2031                 p_attuale[
2032                     1] < (
2033                         list1[1] + 2) and (
2034                             list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
2035                 a = False
2036             else:
2037                 time.sleep(0.1)
2038
2039             state = client.getMultirotorState(vehicle_name)
2040             p_attuale = [state.kinematics_estimated.position.x_val,
2041                 state.kinematics_estimated.position.y_val,
2042                 state.kinematics_estimated.position.z_val]
2043
2044         def landing(self, drone_number):
2045
2046             with self._lock:
2047
2048                 vehicle_name = 'Drone' + str(drone_number + 1)
2049
2050                 client.moveToPositionAsync(x_home, y_home, -3, 3, vehicle_name=vehicle_name)
2051
2052                 client.moveByVelocityZAsync(x_home, y_home, z_home, 2, airsim.DrivetrainType.MaxDegreeOfFreedom,
2053                     airsim.YawMode(False, 0),
2054                     vehicle_name='Drone' + str(drone_number + 1)).join()
2055
2056                 client.armDisarm(False, vehicle_name='Drone' + str(drone_number + 1))
2057
2058                 client.enableApiControl(False, vehicle_name='Drone' + str(drone_number + 1))
2059
2060
2061

```

```
2062 ▶ if __name__ == "__main__":  
2063  
2064     program = MainProgram()  
2065  
2066     with concurrent.futures.ThreadPoolExecutor(max_workers=population) as executor:  
2067         for index in range(population):  
2068             executor.submit(program.execution, index)  
2069  
2070     with concurrent.futures.ThreadPoolExecutor(max_workers=population) as executor:  
2071         for index in range(population):  
2072             executor.submit(program.landing, index)  
2073  
2074
```